Embedded Systems

Multimedia Multiprocessor Systems

Analysis, Design and Management

Akash Kumar · Henk Corporaal
Bart Mesman · Yajun Ha

Springer

Embedded Systems

For other titles published in this series, go to
www.springer.com/series/8563

Akash Kumar • Henk Corporaal •
Bart Mesman • Yajun Ha

Multimedia Multiprocessor Systems

Analysis, Design and Management

Springer

Dr. Akash Kumar
Eindhoven University of Technology
Eindhoven
Netherlands
and
National University of Singapore
Electrical and Computer Engineering
Engineering Drive 3 4
117583 Singapore
Singapore
eleak@nus.edu.sg

Prof. Dr. Henk Corporaal
Eindhoven University of Technology
Electrical Engineering
Den Dolech 2
5612 AZ Eindhoven
Netherlands
h.corporaal@tue.nl

Dr. Bart Mesman
Eindhoven University of Technology
Electrical Engineering
Den Dolech 2
5612 AZ Eindhoven
Netherlands
b.mesman@tue.nl

Asst. Prof. Yajun Ha
National University of Singapore
Electrical and Computer Engineering
Engineering Drive 3 4
117583 Singapore
Singapore
elehy@nus.edu.sg

ISBN 978-94-007-0082-6 e-ISBN 978-94-007-0083-3
DOI 10.1007/978-94-007-0083-3
Springer Dordrecht Heidelberg London New York

Library of Congress Control Number: 2010936448

Cover design: VTEX, Vilnius

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Preface and Outline

Computing systems are around for a relatively short period; it is only since the invention of the microprocessor systems in the early seventies that processors became affordable by everybody. Although this period is short it is hardly imaginable how current life would be without computing systems. They penetrated virtually all aspects of life. Most visible are PCs, smartphones and gaming consoles. Almost every household has several of them, counting up to billions worldwide. However, we are even more dependent on so-called embedded systems; that are computing systems which are usually not visible, but they largely determine the functionality of the surrounding system or equipment. They control factories, take care of security (using e.g. smart cameras), control your car and its engine (actually your car likely contains tens of embedded systems), calculate your travel route, take care of internet traffic, control your coffee machine, etc.; they make the surrounding system behave more intelligently. As a consequence we are dependent on them.

One of the reasons computing systems are everywhere is that they are cheap, and they are cheap because of the ongoing integration and miniaturization. While we needed a large room to install the earliest computing systems, now a processor can be integrated in a few square millimeters or less, while giving substantial performance in terms of operations per second.

Talking about performance, performance demands are constantly increasing. This holds especially for multi-media type of systems, that are systems which process streams of data, data being e.g. audio, all kinds of sensing data, video and graphics. To give an example, the next generation of smartphones is estimated to require about one Tera operations per second for its video, vision, graphics, GPS, audio and speech recognition capabilities, and to make matters worse, this performance has be delivered in a small package and should leave your batteries operational for many days. To deliver this performance for such a small energy budget, computing systems will contain many processors. Some of these processors will be general purpose programmable; others need to be more tuned to the application domain in order to reach the required computational efficiency (in operations

per second per Watt). Future multi-media systems will therefore be heterogeneous multi-core.

Multi-media will not be the sole domain of advanced smartphones and the like. Since cameras are getting very cheap we predict that many future smart systems will be enhanced by adding vision capabilities like surveillance, recognition, virtual reality, visual control, and so on. Multi-media systems can be dedicated for one specific application; however an ongoing trend is to map multiple applications to the same system. This sharing of resources by several applications makes the system cheaper and more versatile, but substantially adds to its design complexity.

Although integration is still following Moore's law, making computing systems cheaper, at least when counting chip area, their design becomes extremely complex, and therefore very costly. This is not only caused by increased functionality and performance demands of the running applications, but also by other, non-functional, requirements like energy cost and real-time demands. Making systems functionally correct is already quite challenging, but making this functionality operating at the right speed, obeying severe throughput and latency requirements can become a nightmare and may lead to many and large debugging and redesign cycles. Therefore a more systematic design method is urgently needed which avoids large debugging cycles while taking real-time demands into account as an integral part of the design process.

Aim of This Book

In this book we focus on (streaming) multi-media systems, and in particular on the real-time aspects of these systems. These systems run multiple applications and are realized using multiple processing cores, some of them specialized for a specific target application domain. The goal of this book is to make you familiar with techniques to model and analyze these systems, both hardware and software, while taking timing requirements into account. The models will be taken from the data flow modeling domain; in particular we will teach you how to use SDF (Synchronous Data Flow) models to specify and analyze your system. Although SDF is restricted in its expressive power, it has very good analysis power, and appropriate tools exist to perform this analysis. For example, timing properties and deadlock avoidance can be easily verified. It also allows for the calculation of appropriate buffer sizes for the inter-core communication buffers. Based on this specification and analysis this book teaches you how to synthesize and implement the specified multi-processor system using FPGA technology. This synthesis is correct by construction, and therefore it avoids many debugging iterations. The FPGA implementation can also act as a quick prototype for a final silicon realization.

Mapping multiple applications to such a system requires a run-time manager. This manager is responsible for admission control, i.e., can a new application be added to the other, already running applications, such that every application still

meets its timing requirements. Once admitted the run-time manager is also responsible for controlling the resources and enforcing the right time budgets to all applications. This book will show several techniques for performing these management tasks.

You may not be satisfied by just running one set of applications. The set of running applications may regularly change, leading to multiple use cases. This adds new dimensions to the design process, especially when mapping to FPGAs; some of them will be treated at the end of this book. E.g. how do you share the resources of an FPGA between the multiple use cases, this to reduce the number of FGPA reconfigurations. Another dimension discussed is the estimation of the amount of FPGA resources needed by a set of use cases prior to the synthesis.

In short you will learn how to map multiple applications, possibly divided into multiple use cases to a multi-processor system, and you will be able to quickly realize such a system into an FPGA. All the theory discussed in this book is supported by a complete design flow called MAMPS, which stands for Multiple Applications Multi-Processor Synthesis. The flow is fully implemented and demonstrated by several examples.

Within the book we make several restrictions. A major one is that we mainly deal with soft real-time constraints. The techniques used in this book do not give hard real-time guarantees (unless indicated otherwise). The reason is that giving hard real-time guarantees may result in severe overestimation of the required resources, and therefore may give a huge performance and cost penalty. This does not mean that we do not emphasize research in hard real-time systems. On the contrary; it is one of our major research themes, already for many years, and the focus of many of our current projects. The reader is referred to the book website (see below) for further information on our other projects.

Audience

This book covers a complete design trajectory for the design of multi-media systems running multiple applications; it includes both theory and practice. It is meant for all people interested in designing multi-media and other real-time multi-processor systems. It helps them to think and reason about timing requirements and offers them various modeling, analysis, design and management techniques needed to realize these complex systems first time right. The book is also meant for system level architects who want to quickly make high level estimates and system trade-offs based on solid modeling. The book is also suitable for use within a post graduate course. To this purpose we included extensive introductory chapters on trends and challenges in multi-media systems, and on the theory behind application modeling and scheduling. In particular data flow techniques are treated in some depth. In such a course the available tools will help students to get familiar with future design flows, and bring the theory into practice.

Accompanying Material

Besides the printed copy, there is an accompanying book website at http://www.es.ele.tue.nl/~akash/MMSBook. This website contains further information about designing real-time systems and various links to other related research. In addition to that, accompanying slides can be found on the website. The slides can be used in a course, provided the copyright information is retained.

As mentioned most of the presented techniques and algorithms are integrated in the MAMPS design flow. The corresponding tooling, and its documentation, can be found at http://www.es.ele.tue.nl/MAMPS. The tools can be used online. The site contains a couple of tested examples to try out the tools. For collaborating partners tools can be made available on request for development.

This book is based on research and development being performed at the TU/e, the Eindhoven University of Technology, within the PreMaDoNA project of the Electronic Systems group. PreMaDoNA stands for predictable matching of demands on networked architectures. Other results from this project and it's follow up projects can also be found following the links on the book website.

Organization of This Book

This book is divided into seven chapters. The first two are introductory. The first one describes trends in multimedia systems; the second one goes into the theory behind data flow modeling and scheduling, and introduces the necessary notation. Chapter 3 describes our new iterative analysis method. Chapter 4 treats how to perform resource management. Chapter 5 describes the MAMPS design flow which allows for quick realization of a system into an FPGA. Chapter 6 extends the system to support multiple use cases. Finally Chap. 7 gives several conclusions and outlines the open problems that are not solved in this book. Although the best way is to read all chapters in the presented order, some readers may find it convenient to skip parts on first reading. Chapters 3, 4 and 5 do not have big interdependences. Therefore, after reading the first 2 chapters, readers can select continue with either Chap. 3, 4 or 5. Chapter 6 depends on Chap. 5, so readers interested in mapping multiple use cases should first read Chap. 5.

Henk Corporaal

Contents

List of Figures

List of Tables

Chapter 1
Trends and Challenges in Multimedia Systems

Odyssey, released by Magnavox in 1972, was the world's first video game console (Odyssey 1972). This supported a variety of games from tennis to baseball. Removable circuit cards consisting of a series of jumpers were used to interconnect different logic and signal generators to produce the desired game logic and screen output components respectively. It did not support sound, but it did come with translucent plastic overlays that one could put on the TV screen to generate colour images. This was what is called as the first generation video game console. Figure 1.1(a) shows a picture of this console, that sold about 330,000 units. Let us now forward to the present day, where the video game consoles have moved into the seventh generation. An example of one such console is the PlayStation3 from Sony (PS3 2006) shown in Fig. 1.1(b), that sold over 21 million units in the first two years of its launch. It not only supports sounds and colours, but is a complete media centre which can play photographs, video games, movies in high definitions in the most advanced formats, and has a large hard-disk to store games and movies. Further, it can connect to one's home network, and the entire world, both wireless and wired. Surely, we have come a long way in the development of multimedia systems.

A lot of progress has been made from both applications and system-design perspective. The designers have a lot more resources at their disposal – more transistors to play with, better and almost completely automated tools to place and route these transistors, and much more memory in the system. However, a number of key challenges exist. With increasing number of transistors has come increased power to worry about. While the tools for the back-end (synthesizing a chip from the detailed system description) are almost completely automated, the front-end (developing a detailed specification of the system) of the design-process is still largely manual, leading to increased design time and error. While the cost of memory in the system has decreased a lot, its speed has little. Further, the demands from the application have increased even further. While the cost of transistors has declined, increased competition is forcing companies to cut cost, in turn forcing designers to use as few resources as necessary. Systems have evolving standards often requiring a complete re-design often late in the design-process. At the same time, the time-to-market is decreasing, making it even harder for the designer to meet the strict deadlines.

A. Kumar et al., *Multimedia Multiprocessor Systems*, Embedded Systems,
DOI 10.1007/978-94-007-0083-3_1,

(a) (b)

Fig. 1.1 Comparison of world's first video console with one of the most modern consoles. (**a**) Odyssey, released in 1972 – an example from first generation video game console (Odyssey 1972). (**b**) Sony PlayStation3 released in 2006 – an example from the seventh generation video game console (PS3 2006)

In this book, we present analysis, design and management techniques for modern multimedia systems which are increasingly using multi-processor platforms to meet various performance requirements. To cope with the complexity in designing such systems, a largely automated design-flow is needed that can generate systems from a high-level system description such that they are not error-prone and consume less time. This book presents a highly automated flow – *MAMPS* (Multi-Application Multi-Processor Synthesis), that synthesizes multi-processor platforms for not just multiple applications, but *multiple use-cases.* (A *use-case* is defined as a combination of applications that may be active concurrently.) One of the key design automation challenges that remains is fast exploration of software and hardware implementation alternatives with accurate performance evaluation. Techniques are presented to merge multiple use-cases into one hardware design to minimize cost and design time, making it well-suited for fast design space exploration of MPSoC systems.

In order to limit the design-cost it is important to have a system that is neither hugely over-dimensioned, nor too limited to support modern applications. While there are techniques to estimate application performance, they often end-up providing a high upper bound such that the hardware is grossly over-dimensioned. We present a performance prediction methodology that can accurately and quickly predict the performance of multiple applications before they execute in the system. The technique is fast enough to be used at run-time as well. This allows run-time addition of applications in the system. An admission controller is presented using the analysis technique that admits incoming applications only if their performance is expected to meet their desired requirements. Further, a mechanism is presented to

manage resources in a system. This ensures that once an application is admitted in the system, it can meet its performance constraints. The entire set-up is integrated in the *MAMPS* flow and available on-line for the benefit of research community.

This chapter is organized as follows. In Sect. 1 we take a closer look at the trends in multimedia systems from the applications perspective. In Sect. 2 we look at the trends in multimedia system design. Section 3 summarizes the key challenges that remain to be solved as seen from the two trends. Section 4 explains the overall design flow that is used in this book. Section 5 gives an overview of the book.

1 Trends in Multimedia Systems Applications

Multimedia systems are systems that use a combination of content forms like text, audio, video, pictures and animation to provide information or entertainment to the user. The video game console is just one example of the many multimedia systems that abound around us. Televisions, mobile phones, home theatre systems, mp3 players, laptops, personal digital assistants, are all examples of multimedia systems. Modern multimedia systems have changed the way in which users receive information and expect to be entertained. Users now expect information to be available instantly whether they are travelling in the airplane, or sitting in the comfort of their houses. In line with users' demand, a large number of multimedia products are available. To satisfy this huge demand, the semiconductor companies are busy releasing newer embedded, and multimedia systems in particular, every few months.

The number of features in a multimedia system is constantly increasing. For example, a mobile phone that was traditionally meant to support voice calls, now provides video-conferencing features and streaming of television programs using 3G networks. An mp3 player, traditionally meant for simply playing music, now stores contacts and appointments, plays photos and video clips, and also doubles up as a video game. Some people refer to it as the convergence of information, communication and entertainment (Baldwin et al. 1996). Devices that were traditionally meant for only one of the three things, now support all of them. The devices have also shrunk, and they are often seen as fashion accessories. A mobile phone that was not very *mobile* until about 15 years ago, is now barely thick enough to support its own structure, and small enough to hide in the smallest of ladies-purses.

Further, many of these applications execute concurrently on the platform in different combinations. We define each such combination of simultaneously active applications as a **use-case**. (It is also known as *scenario* in literature (Paul et al. 2006).) For example, a mobile phone in one instant may be used to talk on the phone while surfing the web and downloading some Java application in the background. In another instant it may be used to listen to mp3 music while browsing JPEG pictures stored in the phone, and at the same time allow a remote device to access the files in the phone over a bluetooth connection. Modern devices are built to support different use-cases, making it possible for users to choose and use the desired functions concurrently.

Another trend we see is increasing and evolving standards. A number of standards for radio communication, audio and video encoding/decoding and interfaces are available. The multimedia systems often support a number of these. While a high-end TV supports a variety of video interfaces like HDMI, DVI, VGA and coaxial cable; a mobile phone supports multiple bands like GSM 850, GSM 900, GSM 180 and GSM 1900, besides other wireless protocols like Infrared and Bluetooth (Magoon et al. 2002; Kahn and Barry 1997; Bluetooth 2004). As standards evolve, allowing faster and more efficient communication, newer devices are released in the market to match those specifications. The time to market is also reducing since a number of companies are in the market (Jerraya and Wolf 2004), and the consumers expect quick releases. A late launch in the market directly hurts the revenue of the company.

Power consumption has become a major design issue since many multimedia systems are hand-held. According to a survey by TNS research, two-thirds of mobile phone and PDA users rate *two-days of battery life during active use* as the most important feature of the ideal converged device of the future (TNS 2006). While the battery life of portable devices has generally been increasing, the active use is still limited to a few hours, and in some extreme cases to a day. Even for other plugged multimedia systems, power has become a global concern with rising oil prices, and a growing general awareness to reduce energy consumption.

To summarize, we see the following trends and requirements in the application of multimedia devices.

- An increasing number of multimedia devices are being brought to market.
- The number of applications in multimedia systems is increasing.
- The diversity of applications is increasing with convergence and multiple standards.
- The applications execute concurrently in varied combinations known as use-cases, and the number of these use-cases is increasing.
- The time-to-market is reducing due to increased competition, and evolving standards and interfaces.
- Power consumption is becoming an increasingly important concern for future multimedia devices.

2 Trends in Multimedia Systems Design

A number of factors are involved in bringing the progress outlined above in multimedia systems. Most of them can be directly or indirectly attributed to the famous Moore's law (Moore 1965), that predicted the exponential increase in transistor density as early as 1965. Since then, almost every measure of the capabilities of digital electronic devices – processing speed, transistor count per chip, memory capacity, even the number and size of pixels in digital cameras – are improving at roughly exponential rates. This has had two-fold impact. While on one hand, the hardware designers have been able to provide bigger, better and faster means of processing,

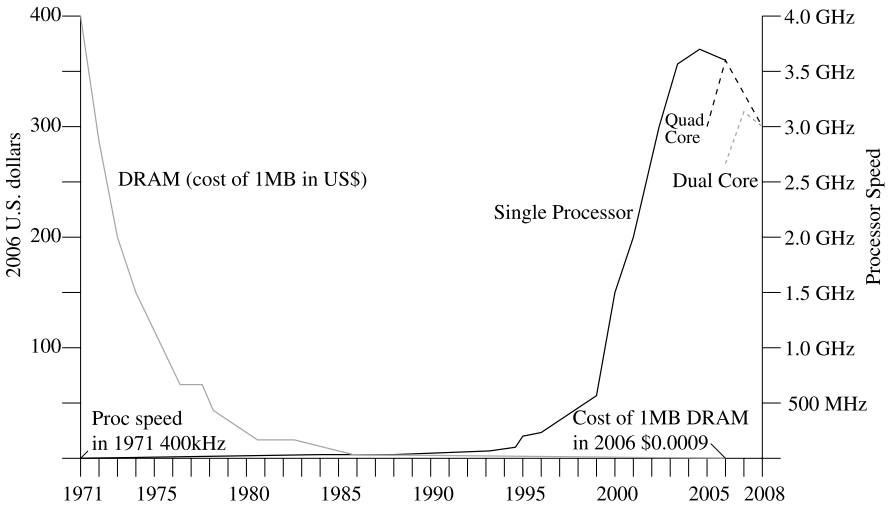
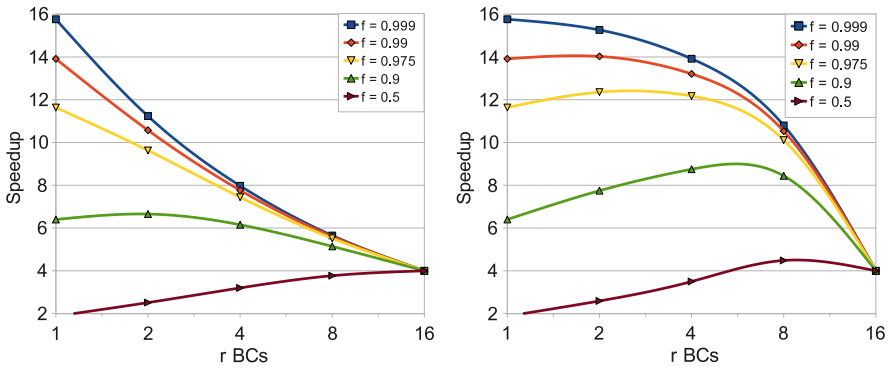


Fig. 1.2 Increasing processor speed and reducing memory cost (Adee 2008)

on the other hand, the application developers have been working hard to utilize this processing power to its maximum. This has led them to deliver better and increasingly complex applications in all dimensions of life – be it medical care systems, airplanes, or multimedia systems.

When the first Intel processor was released in 1971, it had 2,300 transistors and operated at a speed of 400 kHz. In contrast, a modern chip has more than a billion transistors operating at more than 3 GHz (Intel 2004). Figure 1.2 shows the trend in processor speed and the cost of memory (Adee 2008). The cost of memory has come down from close to 400 U.S. dollars in 1971, to less than a cent for 1 MB of dynamic memory (RAM). The processor speed has risen to over 3.5 GHz. Another interesting observation from this figure is the introduction of dual and quad core chips since 2005 onwards. This indicates the beginning of a multi-processor era. As the transistor size shrinks, they can be clocked faster. However, this also leads to an increase in power consumption, in turn making chips hotter. Heat dissipation has become a serious problem forcing chip manufacturers to limit the maximum frequency of the processor. Chip manufacturers are therefore, shifting towards designing multiprocessor chips operating at a lower frequency. Intel reports that *under-clocking* a single core by 20 percent saves half the power while sacrificing just 13 percent of the performance (Ross 2008). This implies that if the work is divided between two processors running at 80 percent clock rate, we get 74 percent better performance for the same power. In addition, the heat is dissipated at two points rather than one.

Further, sources like Berkeley and Intel are already predicting hundreds and thousands of cores on the same chip (Asanovic et al. 2006; Borkar 2007) in the near future. All computing vendors have announced chips with multiple processor cores. Moreover, vendor road-maps promise to repeatedly double the number of cores per chip. These future chips are variously called *chip multiprocessors*, *multi-core chips*,



(a) Homogeneous systems (b) Heterogeneous systems

Fig. 1.3 Comparison of speedup obtained by combining r smaller cores into a bigger core in homogeneous and heterogeneous systems (Hill and Marty 2008)

and *many-core chips*, and the complete system as *multi-processor systems-on-chip* (MPSoC).

Following are the key benefits of using multi-processor systems.

- They consume less power and energy, provided sufficient task-level parallelism is present in the application(s). If there is insufficient parallelism, then some processors can be switched off.
- Multiple applications can be easily shared among processors.
- Streaming applications (typical multimedia applications) can be more easily pipelined.
- More robust against failure – a Cell processor is designed with 8 vector cores (also known as SPE), but not all are always working, e.g. in a PS3 only 7 are guaranteed to work.
- Heterogeneity can be supported, allowing better performance.
- It is more scalable, since higher performance can be obtained by adding more processors.

In order to evaluate the true benefits of multi-core processing, Amdahl's law (Amdahl 1967) has been augmented to deal with multi-core chips (Hill and Marty 2008). Amdahl's law is used to find the maximum expected improvement to an overall system when only a part of the system is improved. It states that if you enhance a fraction f of a computation by a speedup S, the overall speedup is:

$$Speedup_{enhanced}(f, S) = \frac{1}{(1-f) + \frac{f}{S}}$$

However, if the sequential part can be made to execute in less time by using a processor that has better sequential performance, the speedup can be increased. Suppose we can use the resources of r base-cores (BCs) to build one bigger core, which gives a performance (relative to 1 base-core) of $perf(r)$. If $perf(r) > r$ i.e. super linear speedup, it is always advisable to use the bigger core, since doing so speeds up both

sequential and parallel execution. However, usually $perf(r) < r$. When $perf(r) < r$, trade-off starts. Increasing core performance helps in sequential execution, but hurts parallel execution. If resources for n BCs are available on a chip, and all BCs are replaced with n/r bigger cores, the overall speedup is:

$$Speedup_{homogeneous}(f, n, r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f.r}{perf(r).n}}$$

When heterogeneous multiprocessors are considered, there are more possibilities to redistribute the resources on a chip. If only r BCs are replaced with 1 bigger core, the overall speedup is:

$$Speedup_{heterogeneous}(f, n, r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{perf(r)+n-r}}$$

Figure 1.3 shows the speedup obtained for both homogeneous and heterogeneous systems, for different fractions of parallelizable software. The x-axis shows the number of base processors that are combined into one larger core. In total there are resources for 16 BCs. The origin shows the point when we have a homogeneous system with only base-cores. As we move along the x-axis, the number of base-core resources used to make a bigger core are increased. In a homogeneous system, all the cores are replaced by a bigger core, while for heterogeneous, only one bigger core is built. The end-point for the x-axis is when all available resources are replaced with one big core. For this figure, it is assumed that $perf(r) = \sqrt{r}$. As can be seen, the corresponding speedup when using a heterogeneous system is much greater than homogeneous system. While these graphs are shown for only 16 base-cores, similar performance speedups are obtained for other bigger chips as well. This shows that using a heterogeneous system with several large cores on a chip can offer better speedup than a homogeneous system.

In terms of power as well, heterogeneous systems are better. Figure 1.4 shows the *intrinsic computational efficiency* of silicon as compared to that of microprocessors (Roza 2001). The graph shows that the flexibility of general purpose microprocessors comes at the cost of increased power. The upper staircase-like line of the figure shows Intrinsic Computational Efficiency (ICE) of silicon according to an analytical model from (Roza 2001) ($MOPS/W \approx \alpha/\lambda V_{DD}^2$, α is constant, λ is feature size, and V_{DD} is the supply voltage). The intrinsic efficiency is in theory bounded by the number of 32-bit mega (adder) operations that can be achieved per second per Watt. The performance discontinuities in the upper staircase-like line are caused by changes in the supply voltage from 5 V to 3.3 V, 3.3 V to 1.5 V and 1.5 V to 1.2 V. We observe that there is a gap of 2-to-3 orders of magnitude between the intrinsic efficiency of silicon and general purpose microprocessors. The accelerators – custom hardware modules designed for a specific task – come close to the maximum efficiency. Clearly, it may not always be desirable to actually design a hypothetically maximum efficiency processor. A full match between the application and architecture can bring the efficiency close to the hypothetical maximum. A heterogeneous platform may combine the flexibility of using a general purpose microprocessor

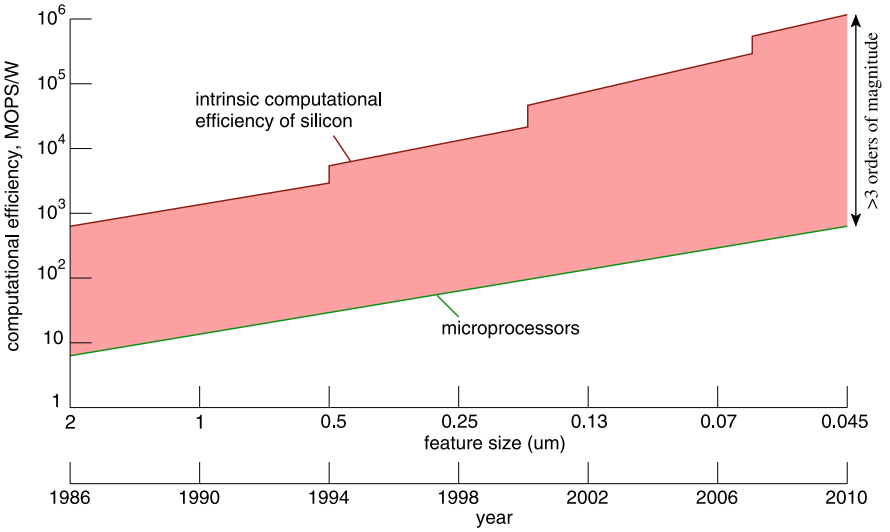


Fig. 1.4 The intrinsic computational efficiency of silicon as compared to the efficiency of microprocessors

and custom accelerators for compute intensive tasks, thereby minimizing the power consumed in the system.

Most modern multiprocessor systems are heterogeneous, and contain one or more application-specific processing elements (PEs). The CELL processor (Kahle et al. 2005), jointly developed by Sony, Toshiba and IBM, contains up to nine PEs – one general purpose PowerPC (Weiss and Smith 1994) and eight *Synergistic Processor Elements (SPEs)*. The PowerPC runs the operating system and the control tasks, while the SPEs perform the compute-intensive tasks. This Cell processor is used in PlayStation3 described above. STMicroelectronics Nomadik contains an ARM processor and several *Very Long Instruction Word (VLIW)* DSP cores (Artieri et al. 2003). Texas Instruments OMAP processor (Cumming 2003), DaVinci processor (Texas Instruments 2008) and Philips Nexperia (De Oliveira and Van Antwerpen 2003) are other examples. Recently, many companies have begun providing configurable cores that are targeted towards an application domain. These are known as *Application Specific Instruction-set Processors (ASIPs)*. These provide a good compromise between general-purpose cores and ASICs. Tensilica (Tensilica 2009; Gonzalez 2000) and Silicon Hive (Hive 2004; Halfhill 2005) are two such examples, which provide the complete toolset to generate multiprocessor systems where each processor can be customized towards a particular task or domain, and the corresponding software programming toolset is automatically generated for them. This also allows the re-use of IP (Intellectual Property) modules designed for a particular domain or task.

Another trend that we see in multimedia systems design is the use of *Platform-Based Design* paradigm (Sangiovanni-Vincentelli et al. 2004; Keutzer et al. 2000). This is becoming increasingly popular due to three main factors: (1) the dramatic

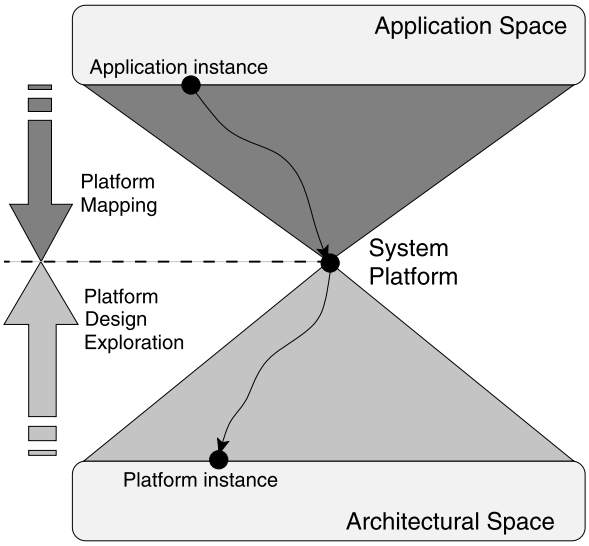


Fig. 1.5 Platform-based design approach – system platform stack

increase in non-recurring engineering cost due to mask making at the circuit implementation level, (2) the reducing time to market, and (3) streamlining of industry – chip fabrication and system design, for example, are done in different companies and places. This paradigm is based on segregation between the system design process, and the system implementation process. The basic tenets of platform-based design are identification of design as *meeting-in-the-middle process*, where successive refinements of specifications meet with abstractions of potential implementations, and the identification of precisely defined abstraction layers where the refinement to the subsequent layer and abstraction processes take place (Sangiovanni-Vincentelli et al. 2004). Each layer supports a design stage providing an opaque abstraction of lower layers that allows accurate performance estimations. This information is incorporated in appropriate parameters that annotate design choices at the present layer of abstraction. These layers of abstraction are called platforms. For MPSoC system design, this translates into abstraction between the application space and architectural space that is provided by the system-platform. Figure 1.5 captures this system-platform that decouples the application development process from the architecture implementation process.

We further observe that for high-performance multimedia systems (like cell-processing engine), non-preemptive systems are preferred over preemptive ones for a number of reasons (Jeffay et al. 1991). In many practical systems, properties of device hardware and software either make the preemption impossible or prohibitively expensive due to extra hardware and (potential) execution time needed. Further, non-preemptive scheduling algorithms are easier to implement than preemptive algorithms and have dramatically lower overhead at run-time (Jeffay et al. 1991). Even in multi-processor systems with preemptive processors, some processors (or co-processors/accelerators) are usually non-preemptive; for such processors

non-preemptive analysis is still needed. It is therefore important to investigate non-preemptive multi-processor systems.

To summarize, the following trends can be seen in the design of multimedia systems.

- *Increase in system resources:* The resources available for disposal in terms of processing and memory are increasing exponentially.
- *Use of multiprocessor systems:* Multi-processor systems are being developed for reasons of power, efficiency, robustness, and scalability.
- *Increasing heterogeneity:* With the re-use of IP modules and design of custom (co-)processors (ASIPs), heterogeneity in MPSoCs is increasing. Heterogeneity also enables high computational efficiency.
- *Platform-based design:* Platform-based design methodology is being employed to improve the re-use of components and shorten the development cycle.
- *Non-preemptive processors:* Non-preemptive processors are preferred over pre-emptive to reduce cost.

3 Key Challenges in Multimedia Systems Design

The trends outlined in the previous two sections indicate the increasing complexity of modern multimedia systems. They have to support a number of concurrently executing applications with diverse resource and performance requirements. The designers face the challenge of designing such systems at low cost and in short time. In order to keep the costs low, a number of design options have to be explored to find the optimal or near-optimal solution. The performance of applications executing on the system have to be carefully evaluated to satisfy user-experience. Run-time mechanisms are needed to deal with run-time addition of applications. In short, following are the major challenges that remain in the design of modern multimedia systems, and are addressed in this book.

- *Multiple use-cases:* Analyzing performance of multiple applications executing concurrently on heterogeneous multi-processor platforms. Further, this number of use-cases and their combinations is exponential in the number of applications present in the system making it computationally infeasible to analyse performance of all use-cases. (*Analysis and Design*)
- *Design and Program:* Systematic way to design *and* program multi-processor platforms. (*Design*)
- *Design space exploration:* Fast design space exploration technique. (*Analysis and Design*)
- *Run-time addition of applications:* Deal with run-time addition of applications – keep the analysis fast and scalable, adapt the design (-process), manage the resources at run-time (e.g. admission controller). (*Analysis, Design and Management*)
- *Meeting performance constraints:* A good mechanism for keeping performance of all applications executing above the desired level. (*Design and Management*)

Each of the above challenges deals with one or more aspects of analysis, design and management as shown above. Each of the three aspects will be introduced in detail now.

Analysis

This book presents a novel probabilistic performance prediction (P^3) algorithm for predicting performance of multiple applications executing on multi-processor platforms. The algorithm predicts the time that tasks have to spend during contention phase for a resource. The computation of accurate waiting time is the key to performance analysis. When applications are modeled as synchronous dataflow (SDF) graphs – a model of computation, their performance on a (multi-processor) system can be easily computed when they are executing *in isolation* (provided we have a *good* model). When they execute concurrently, depending on whether the used scheduler is static or dynamic, the arbitration on a resource is either fixed at design-time or chosen at run-time respectively (explained in more detail in Chap. 2). In the former case, the execution order can be modeled in the graph, and the performance of the entire application can be determined. The contention is therefore modeled as dependency edges in the SDF graph. However, this is more suited for static applications. For dynamic applications such as multimedia, a dynamic scheduler is more suitable. For dynamic scheduling approaches, the contention has to be modeled as waiting time for a task, which is added to the execution time to give the total response time. The performance can be determined by computing the performance (throughput) of this resulting SDF graph. With lack of good techniques for accurately predicting the time spent in contention, designers have to resort to worst-case waiting time estimates, that lead to over-designing the system and loss of performance. Further, those approaches are not scalable and the over-estimate increases with the number of applications.

In this book, we present a solution to performance prediction, with easy analysis. We highlight the issue of *composability* i.e. mapping and analysis of performance of multiple applications on a multiprocessor platform in isolation, as far as possible. This limits computational complexity and allows high dynamism in the system. While in this book, we only show examples with processor contention, memory and network contention can also be easily modeled in SDF graph as shown in (Stuijk 2007). The technique presented here can therefore be easily extended to other system components as well. The analysis technique can be used both at design-time and run-time.

We would ideally want to analyze each application in isolation, thereby reducing the analysis time to become linear in the number of running applications, and still reason about the overall behaviour of the system. One way to achieve this, would be complete *virtualization*. This essentially implies dividing the available resources by the total number of applications in the system. The application would then have exclusive access to its share of resources. For example, if we have 100 MHz processors and a total of 10 applications in the system, each application would get 10 MHz

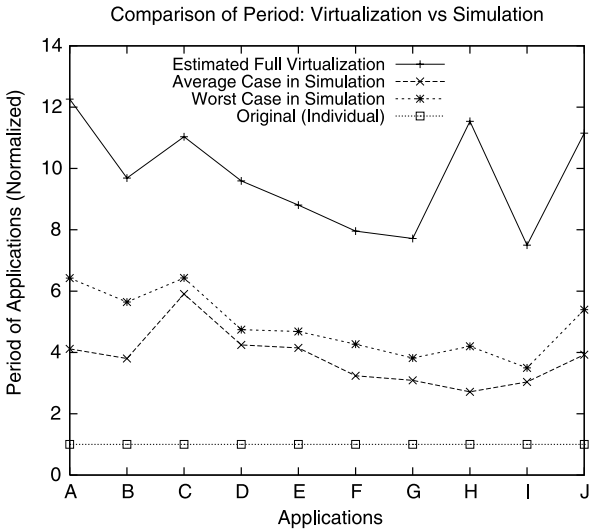


Fig. 1.6 Application performance as obtained with full virtualization in comparison to simulation

of processing resource. The same can be done for communication bandwidth and memory requirements. However this gives two main problems. When fewer than 10 tasks are active, the tasks will not be able to exploit the extra available processing power, leading to wastage. Secondly, the system would be grossly over-dimensioned when the peak requirements of each application are taken into account, even though these peak requirements of applications may rarely occur and never overlap with each other.

Figure 1.6 shows this disparity in more detail. The graph shows the period of ten streaming multimedia applications (inverse of throughput) when they are run concurrently. The period is the time taken for one iteration of the application. The period has been normalized to the original period that is achieved when each application is running in isolation. If full virtualization is used, the period of applications increases to about ten times on average. However, without virtualization, it increases only about five times. A system which is built with full-virtualization in mind, would therefore, utilize only 50% of the resources. Thus, throughput decreases with complete virtualization.

Therefore, a good analysis methodology for a modern multimedia system

- provides accurate performance results, such that the system is not over-dimensioned,
- is fast in order to make it usable for run-time analysis, and to explore a large number of design-points quickly, and
- easily handles a large number of applications, and is composable to allow run-time addition of new applications.

It should be mentioned that often in applications, we are concerned with the long-term throughput and not the individual deadlines. For example, in the case of JPEG application, we are not concerned with decoding of each macro-block, but the

whole image. When browsing the web, individual JPEG images are not as important as the entire page being ready. Thus, for the scope of this book, we consider long-term throughput i.e. cumulative deadline for a large number of iterations, and not just one. However, having said that, it is possible to adapt the analysis to individual deadlines as well. It should be noted that in such cases, the estimates for individual iteration may be very pessimistic as compared to long-term throughput estimates.

Design

As is motivated earlier, modern systems need to support many different combinations of applications – each combination is defined as a *use-case* – on the same hardware. With reducing time-to-market, designers are faced with the challenge of designing and testing systems for multiple use-cases quickly. Rapid prototyping has become very important to easily evaluate design alternatives, and to explore hardware and software alternatives quickly. Unfortunately, lack of automated techniques and tools implies that most work is done by hand, making the design-process error-prone and time-consuming. This also limits the number of design-points that can be explored. While some efforts have been made to automate the flow and raise the abstraction level, these are still limited to single-application designs (or a fixed set of applications).

Modern multimedia systems support not just multiple applications, but also multiple use-cases. The number of such potential use-cases is exponential in the number of applications that are present in the system. The high demand of functionalities in such devices is leading to an increasing shift towards developing systems in software and programmable hardware in order to increase design flexibility. However, a single configuration of this programmable hardware may not be able to support this large number of use-cases with low cost and power. We envision that future complex embedded systems will be partitioned into several configurations and the appropriate configuration will be loaded into the *reconfigurable platform* (defined as a piece of hardware that can be configured at run-time to achieve the desired functionality) on the fly as and when the use-cases are requested. This requires two major developments at the research front: (1) a systematic design methodology for allowing multiple use-cases to be merged on a single hardware configuration, and (2) a mechanism to keep the number of hardware configurations as small as possible. More hardware configurations imply a higher cost since the configurations have to be stored in the memory, and also lead to increased switching within the system.

In this book, we present *MAMPS* (Multi-Application Multi-Processor Synthesis) – a design-flow that generates the entire MPSoC for multiple use-cases from application(s) specifications, together with corresponding software projects for automated synthesis. This allows the designers to quickly traverse the design-space and evaluate the performance on real hardware. Multiple use-cases of applications are supported by merging them in such a way that minimal hardware is generated. This further reduces the time spent in system-synthesis. When not all use-cases can

be supported with one configuration, due to the hardware constraints, multiple configurations of hardware are automatically generated, while keeping the number of partitions low. Further, an area estimation technique is provided that can accurately predict the area of a design and decide whether a given system-design is feasible within the hardware constraints or not. This helps in quick evaluation of designs, thereby making the DSE faster.

Thus, the design-flow presented in this book is unique in a number of ways: (1) it supports multiple use-cases on one programmable hardware platform, (2) estimates the area of design before the actual synthesis, allowing the designer to choose the right device, (3) merges and partitions the use-cases to minimize the number of hardware configurations, and (4) it allows fast DSE by automating the design generation and exploration process.

Management

Resource management, i.e. managing all the resources present in a multiprocessor system, is similar to the task of an operating system on a general purpose computer. This includes starting up of applications, and allocating resources to them appropriately. In the case of a multimedia system (or embedded systems, in general), a key difference from a general purpose computer is that the applications (or application domain) is generally known, and the system can be optimized for them. Further, most decisions can be already taken at design-time to save the cost at run-time. Still, a complete design-time analysis is becoming increasingly harder due to three major reasons: (1) little may be known at design-time about the applications that need to be used in future, e.g. a navigation application like Tom-Tom may be installed on the phone after-wards, (2) the precise platform may also not be known at design time, e.g. some cores may fail at run-time, and (3) the number of design-points that need to be evaluated is prohibitively large. A run-time approach can benefit from the fact that the exact application mix is known, but the analysis has to be fast enough to make it feasible.

In this book, we present a **hybrid approach** for designing systems with multiple applications. This splits the management tasks into off-line and on-line parts. The time-consuming application specific computations are done at design-time and for each application independent from other applications, and the use-case specific computations are performed at run-time. The off-line computation includes things like application-partitioning, application-modeling, determining the task execution times, determining their maximum throughput, etc. Further, parametric equations are derived that allow throughput computation of tasks with varying execution times. All this analysis is time-consuming and best carried out at design-time. Further, in this part no information is needed from the other applications and it can be performed in isolation. This information is sufficient enough to let a run-time manager determine the performance of an application when executing concurrently on the platform with other applications. This allows easy **addition of applications at run-time**. As long as all the properties needed by the run-time resource manager are

derived for the new application, the application can be treated as all the other applications that are present in the system.

At run-time, when the resource manager needs to decide, for example, which resources to allocate to an incoming application, it can evaluate the performance of applications with different allocations and determine the best option. In some cases, multiple quality levels of an application may be specified, and at run-time the resource manager can choose from one of those levels. This functionality of the resource manager is referred to as **admission control**. The manager also needs to ensure that applications that are admitted do not take more resources than allocated, and starve the other applications executing in the system. This functionality is referred to as **budget enforcement**. The manager periodically checks the performance of all applications, and when an application does better than the required level, it is suspended to ensure that it does not take more resources than needed. For the scope of this book, the effect of task migration is not considered since it is orthogonal to our approach.

4 Design Flow

Figure 1.7 shows the design-flow that is proposed in this book. Specifications of applications are provided to the designer in the form of Synchronous Dataflow (SDF) graphs (Sriram and Bhattacharyya 2000; Lee and Messerschmitt 1987). These are often used for modeling multimedia applications. This is further explained in Chap. 2. As motivated earlier in the chapter, modern multimedia systems support a number of applications in varied combinations defined as **use-case**. Figure 1.7 shows three example applications – A, B and C, and three use-cases with their combinations. For example, in *Use-case 2* applications A and B execute concurrently. For each of these use-cases, the performance of all active applications is analyzed. When a suitable mapping to hardware is to be explored, this step is often repeated with different mappings, until the desired performance is obtained. A probabilistic mechanism is used to estimate the average performance of applications. This **performance analysis** technique is presented in Chap. 3.

When a satisfactory mapping is obtained, the system can be designed and synthesized automatically using the **system-design** approach presented in Chap. 5. **Multiple use-cases** need to be merged on to one hardware design such that a new hardware configuration is not needed for every use-case. This is explained in Chap. 6. When it is not possible to merge all use-cases due to resource constraints (slices in an FPGA, for example), use-cases need to be partitioned such that the number of hardware partitions are kept to a minimum. Further, a fast area estimation method is needed that can quickly identify whether a set of use-cases can be merged due to hardware constraints. Trying synthesis for every use-case combination is too time-consuming. A novel area-estimation technique is needed that can save precious time during design space exploration. This is explained in Chap. 6.

Once the system is designed, a run-time mechanism is needed to ensure that all applications can meet their performance requirements. This is accomplished by us-

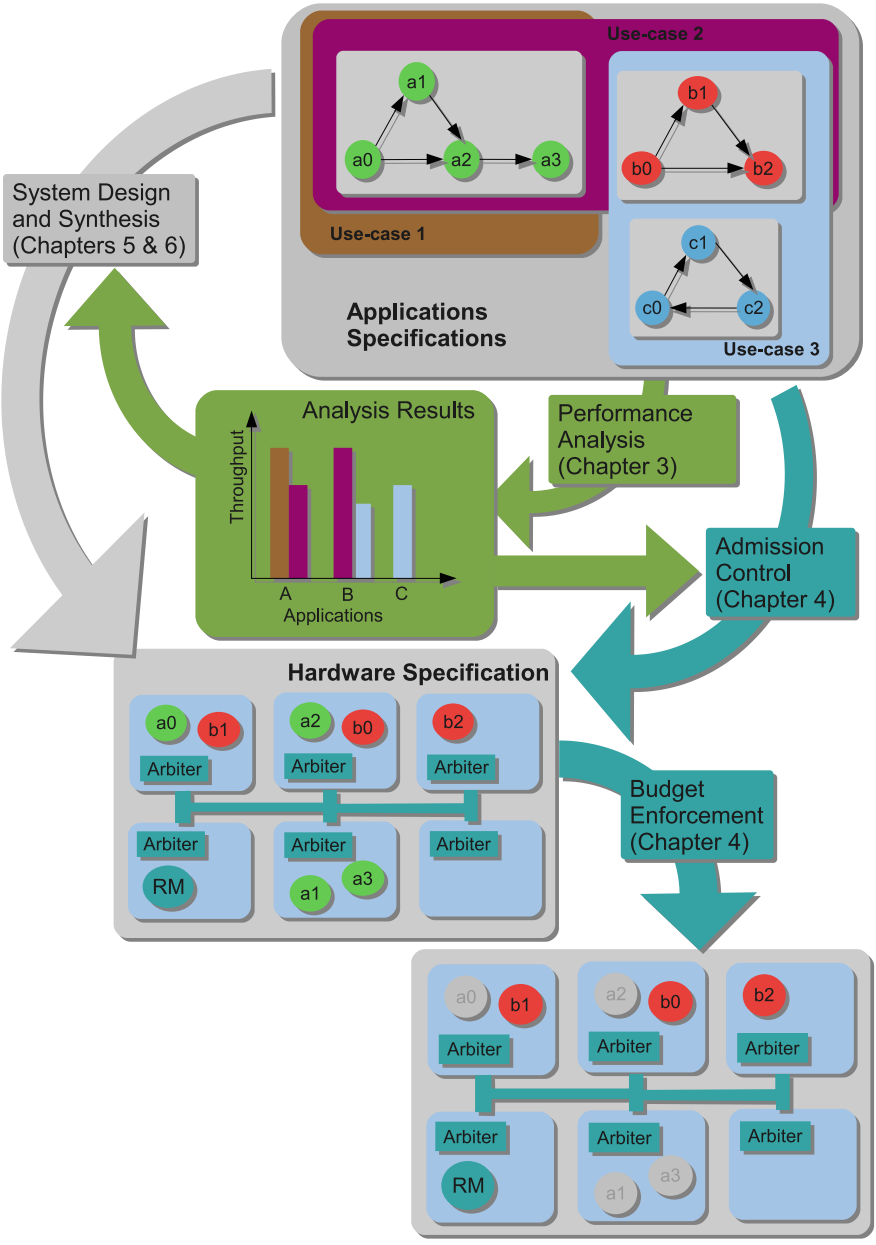


Fig. 1.7 Complete design flow starting from applications specifications and ending with a working hardware prototype on an FPGA

ing a resource manager (RM). Whenever a new application is to be started, the manager checks whether sufficient resources are available. This is defined as **admission-control**. The probabilistic analysis is used to predict the performance of applications when the new application is admitted in the system. If the expected performance of all applications is above the minimum desired performance then the application is started, else a lower quality of incoming application is tried. The resource manager also takes care of **budget-enforcement** i.e. ensuring applications use only as much resources as assigned. If an application uses more resources than needed and starves other applications, it is suspended. Figure 1.7 shows an example where application *A* is suspended. Chapter 4 provides details of two main tasks of the RM – admission control and budget-enforcement.

The above flow also allows for run-time addition of applications. Since the performance analysis presented is fast, it is done at run-time. Therefore, any application whose properties have been derived off-line can be used, if there are enough resources present in the system. This is explained in more detail in Chap. 4.

A tool-flow based on the above for Xilinx FPGAs that is also made available for use on-line for the benefit of research community. This tool is available on-line at www.es.ele.tue.nl/mamps/ (MAMPS 2009).

5 Book Overview

This book is organized as follows. Chapter 2 explains the concepts involved in modeling and scheduling of applications. It explores the problems encountered when analyzing multiple applications executing on a multi-processor platform. The challenge of *Composability*, i.e. being able to analyze applications in isolation with other applications, is presented in this chapter. Chapter 3 presents a performance prediction methodology that can accurately predict the performance of applications at run-time before they execute in the system. A run-time iterative probabilistic analysis is used to estimate the time spent by tasks during contention phase, and thereby predict the performance of applications. Chapter 4 explains the concepts of resource management and enforcing budgets to meet the performance requirements. The performance prediction is used for admission control – one of the main functions of the resource manager. Chapter 5 proposes an automated design methodology to generate program MPSoC hardware designs in a systematic and automated way for multiple applications named *MAMPS*. Chapter 6 explains how systems should be designed when multiple use-cases have to be supported. Algorithms for merging and partitioning use-cases are presented in this chapter as well. Finally, Chap. 7 concludes this book and gives directions for future work.

Chapter 2
Application Modeling and Scheduling

Multimedia applications are becoming increasingly more complex and computation hungry to match consumer demands. If we take video, for example, televisions from leading companies are available with high-definition (HD) video resolution of 1080 × 1920 i.e. more than 2 million pixels (Sony 2009; Samsung 2009; Philips 2009) for consumers and even higher resolutions up to quad HD are showcased in electronic shows (CES 2009). Producing images for such a high resolution is already taxing for even high-end MPSoC platforms. The problem is compounded by the extra dimension of multiple applications sharing the same resources. *Good modeling* is essential for two main reasons: (1) to predict the behaviour of applications on a given hardware without actually synthesizing the system, and (2) to synthesize the system after a feasible solution has been identified from the analysis. In this chapter we will see in detail the model requirements we have for designing and analyzing multimedia systems. We discuss various models of computation, in particular so called dataflow models, and choose one that meets our design-requirements.

Another factor that plays an important role in multi-application analysis is determining when and where a part of application is to be executed, also known as *scheduling*. Heuristics and algorithms for scheduling are called *schedulers*. Studying schedulers is essential for good system design and analysis. In this chapter, we discuss the various types of schedulers for dataflow models. When considering multiple applications executing on multi-processor platforms, three main things need to be taken care of: (1) *assignment* – deciding which task of application has to be executed on which processor, (2) *ordering* – determining the order of task-execution, and (3) *timing* – determining the precise time of task-execution. (Some people also define only ordering and timing as scheduling, and assignment as *binding* or *mapping*.) Each of these three tasks can be done at either compile-time or run-time. In this chapter, we classify the schedulers on this criteria and highlight two of them most suited for use in multiprocessor multimedia platforms. We highlight the issue of *composability,* i.e. mapping and analysis of performance of multiple applications on a multiprocessor platform in isolation, as far as possible. This limits computational complexity and allows high dynamism in the system.

This chapter is organized as follows. The first section motivates the need of modeling applications and the requirements for such a model. Section 2 gives an intro-

A. Kumar et al., *Multimedia Multiprocessor Systems*, Embedded Systems,
DOI 10.1007/978-94-007-0083-3_2,

duction to the synchronous dataflow (SDF) graphs that we use in our analysis. Some properties that are relevant for this book are also explained in the same section. Section 3 discusses several other models of computation (MoCs) that are available, and motivates the choice of SDF graphs as the MoC for our applications. Section 4 discusses state-of-the-art techniques used for estimating performance of applications modeled as SDF graphs. Section 5 provides background on the scheduling techniques used for dataflow graphs in general. Section 6 extends the performance analysis techniques to include hardware constraints as well. Section 7 discusses the issue of Composability in depth. Section 8 provides a comparison between static and dynamic ordering schedulers, and Sect. 9 concludes this chapter.

1 Application Model and Specification

Multimedia applications are often also referred to as **streaming applications** owing to their repetitive nature of execution. Most applications execute for a very long time in a fixed execution pattern. When watching television for example, the video decoding process potentially goes on decoding for hours – an hour is equivalent to 180,000 video frames at a modest rate of 50 frames per second (fps). High-end televisions often provide a refresh rate of even 100 fps, and the trend indicates further increase of this rate. The same goes for an audio stream that usually accompanies the video. The platform has to process continuously to get this output to the user.

In order to ensure that this high performance can be met by the platform, the designer has to be able to model the application requirements. In the absence of a good model, it is very difficult to know in advance whether the application performance can be met at all times, and extensive simulation and testing is needed. Even now, companies report a large effort being spent on verifying the timing requirements of the applications. With multiple applications executing on multiple processors, the potential number of use-cases increases rapidly, and so does the cost of verification.

We start by defining a use-case.

Definition 1 (Use-case) Given a set of n applications $A_0, A_1, \ldots, A_{n-1}$, a use-case U is defined as a vector of n elements $(x_0, x_1, \ldots x_{n-1})$ where $x_i \in \{0, 1\}\ \forall i = 0, 1, \ldots n-1$, such that $x_i = 1$ implies application A_i is active, and $x_i = 0$ implies application A_i is inactive.

In other words, a use-case represents a collection of multiple applications that are active simultaneously. It is often impossible to test a system with all potential input cases in advance. Modern multimedia platforms (high-end mobile phones, for example) allow users to download applications at run-time. Testing for those applications at design-time is simply not possible. A good model of an application can allow for such analysis at run-time.

One of the major challenges that arise when mapping an application to an MPSoC platform is dividing the application load over multiple processors. Two ways are available to parallelize the application and divide the load over more than one

processor, namely task-level parallelism and data-level parallelism. In the former, each processor gets a different part of an application to process, while in the latter, processors operate on the same functionality of application, but different data. For example, in case of JPEG image decoding, inverse discrete cosine transform (IDCT) and colour conversion (CC), among other tasks, need to be performed for all parts (macro-blocks) of an image. Splitting the task of IDCT and CC on different processors is an example of task-level parallelism. Splitting the data, in this case macro-blocks, to different processors is an example of data-level parallelism. To an extent, these approaches are orthogonal and can be applied in isolation or in combination. In this book, we shall focus primarily on task-level parallelism since it is more commonly done for heterogeneous platforms. Data-level parallelism is more suited to homogeneous systems.

Parallelizing an application to make it suitable for execution on a multi-processor platform can be a very difficult task. Whether an application is written from start in a manner that is suitable for a model of computation, or whether the model is extracted from the existing (sequential) application, in either case we need to know how long the execution of each program segment will take, how much data and program memory will be needed for it, and when communication program segments are mapped on different processors, how much communication buffer capacity do we need. Further, we also want to know what is the maximum performance that the application can achieve on a given platform, especially when sharing the platform with other applications. For this, we have to also be able to model and analyze scheduling decisions.

To summarize, following are our requirements from an application model that allow mapping and analysis on a multiprocessor platform:

- *Analyze computational requirements:* When designing an application for MPSoC platform, it is important to know how much computational resource an application needs. This allows the designers to dimension the hardware appropriately. Further, this is also needed to compute the performance estimates of the application as a whole. While sometimes, average case analysis of requirements may suffice, often we also need the worst case estimates, for example in case of real-time embedded systems.
- *Analyze memory requirements:* This constraint becomes increasingly more important as the memory cost on a chip goes high. A model that allows accurate analysis of memory needed for the program execution can allow a designer to distribute the memory across processors appropriately and also determine proper mapping on the hardware.
- *Analyze communication requirements:* The buffer capacity between the communicating tasks (potentially) affects the overall application performance. A model that allows computing these buffer-throughput trade-offs can let the designer allocate appropriate memory for the channel and predict throughput.
- *Model and analyze scheduling:* When we have multiple applications sharing processors, scheduling becomes one of the major challenges. A model that allows us to analyze the effect of scheduling on performance of application(s) is needed.

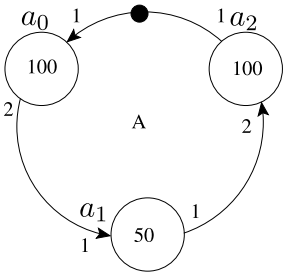


Fig. 2.1 Example of an SDF Graph

- *Design the system:* Once the performance of system is considered satisfactory, the system has to be synthesized such that the properties analyzed are still valid.

Dataflow models of computation fit rather well with the above requirements. They provide a model for describing signal processing systems where infinite streams of data are incrementally transformed by processes executing in sequence or parallel. In a dataflow model, processes communicate via unbounded FIFO channels. Processes read and write atomic data elements or tokens from and to channels. Writing to a channel is non-blocking, i.e. it always succeeds and does not stall the process, while reading from a channel is blocking, i.e. a process that reads from an empty channel will stall and can only continue when the channel contains sufficient tokens. In this book, we use synchronous dataflow (SDF) graphs to model applications and the next section explains them in more detail.

2 Introduction to SDF Graphs

Synchronous Data Flow Graphs (SDFGs, see (Lee and Messerschmitt 1987)) are often used for modeling modern DSP applications (Sriram and Bhattacharyya 2000) and for designing concurrent multimedia applications implemented on multiprocessor systems-on-chip. Both pipelined streaming and cyclic dependencies between tasks can be easily modeled in SDFGs. Tasks are modeled by the vertices of an SDFG, which are called *actors*. The communication between actors is represented by *edges*. Edges represent *channels* for communication in a real system.

The time that the actor takes to execute on a processor is indicated by the number inside the actor. It should be noted that the time an actor takes to execute may vary with the processor. For sake of simplicity, we shall omit the detail as to which processor it is mapped, and just define the time (or clock cycles) needed on a RISC processor (Patterson and Ditzel 1980), unless otherwise mentioned. This is also sometimes referred to as *Timed SDF* in literature (Stuijk 2007). Further, when we refer to the time needed to execute a particular actor, we refer to the worst-case execution-time (WCET). The average execution time may be lower.

Figure 2.1 shows an example SDF graph. There are three actors in this graph. As in a typical data flow graph, a directed edge represents the dependency between actors. Actors need some input data (or control information) before they can start,

and usually also produce some output data; such information is referred to as *tokens*. The number of tokens produced or consumed in one execution of an actor is called *rate*. In the example, a_0 has an input rate of 1 and output rate of 2. Further, its execution time is 100 clock cycles. Actor execution is also called *firing*. An actor is called *ready* when it has sufficient input tokens on all its input edges and sufficient buffer space on all its output channels; an actor can only fire when it is ready.

The edges may also contain *initial tokens*, indicated by bullets on the edges, as seen on the edge from actor a_2 to a_0 in Fig. 2.1. In the above example, only a_0 can start execution from the initial state, since the required number of tokens are present on its single incoming edge. Once a_0 has finished execution, it will produce 2 tokens on the edge to a_1. a_1 can then proceed, as it has enough tokens, and upon completion produce 1 token on the edge to a_2. However, a_2 has to wait before two executions of a_1 are completed, since it needs two input tokens.

Formally, an SDF graph is defined as follows. We assume a set *Ports* of ports, and with each port $p \in Ports$ we associate a finite rate $Rate(p) \in \mathbb{N} \setminus \{0\}$.

Definition 2 (Actor) An actor a is a tuple (In, Out, τ) consisting of a set $In \subseteq Ports$ of input ports (denoted by $In(a)$), a set $Out \subseteq Ports$ of output ports with $In \cap Out = \phi$ and $\tau \in \mathbb{N} \setminus \{0\}$ representing the execution time of a $(\tau(a))$.

Definition 3 (SDF Graph) An SDF graph is a tuple (A, C) consisting of a finite set A of actors and a finite set $C \subseteq Ports^2$ of channels. The channel source is an output port of some actor, the destination is an input port of some actor. All ports of all actors are connected to precisely one channel, and all channels are connected to ports of some actor. For every actor $a = (I, O, \tau) \in A$, we denote the set of all channels that are connected to the ports in I (O) by $InC(a)$ $(OutC(a))$.

When an actor a starts its firing, it removes $Rate(q)$ tokens from all $(p, q) \in InC(a)$. The execution continues for $\tau(a)$ time units and when it ends, it produces $Rate(p)$ tokens on every $(p, q) \in OutC(a)$.

A number of properties of an application can be analyzed from its SDF model. We can calculate the maximum performance possible of an application. We can identify whether the application or a particular schedule will result in a deadlock. We can also analyze other performance properties, e.g. latency of an application, buffer requirements. Below we give some properties of SDF graphs that allow modeling of hardware constraints that are relevant to this book.

Modeling Auto-concurrency

The example in Fig. 2.1 brings a very interesting fact to notice. According to the model, since a_1 requires only one token on the edge from a_0 to fire, as soon as a_0 has finished executing and produced two tokens, two executions of a_1 can start si-

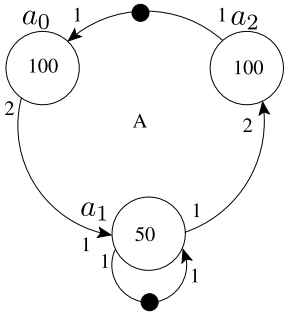


Fig. 2.2 SDF Graph after modeling auto-concurrency of 1 for the actor a_1

multaneously. However, this is only possible if a_1 is mapped and allowed to execute on multiple processors or hardware components simultaneously. In a typical system, a_1 will be mapped on a processor. Once the processor starts executing, it will not be available to start the second execution of a_1 until it has at least finished the first execution of a_1. If there are other actors mapped on it, the second execution of a_1 may even be delayed further.

Fortunately, there is a way to model this particular resource conflict in SDF. Figure 2.2 shows the same example, now updated with the constraint that only one execution of a_1 can be active at any point in time. In this figure, a *self-edge* has been added to actor a_1 with one initial token. For a self-edge, the source and destination actor is the same. This initial token is consumed in the first firing of a_1 and produced after a_1 has finished the first execution. Interestingly enough, by varying the number of initial tokens on this self-edge, we can regulate the number of simultaneous executions of a particular actor. This property is called **auto-concurrency**.

Definition 4 (Auto-concurrency) The **auto-concurrency** of an actor is defined as the maximum number of simultaneous executions of that actor.

In Fig. 2.2, the auto-concurrency of a_1 is 1, while for a_0 and a_2 it is infinite. In other words, the resource conflict for actors a_0 and a_2 is not modeled. In fact, the single initial token on the edge from a_2 to a_0 limits the auto-concurrency of these two actors to one; a self-edge in this case would be superfluous.

Modeling Buffer Sizes

One of the very useful properties of SDF graphs is its ability to model available buffers easily. Buffer-sizes may be modeled as a back-edge with initial tokens. In such cases, the number of tokens on that edge indicates the buffer-size available. When an actor writes data on a channel, the available size reduces; when the receiving actor consumes this data, the available buffer increases, modeled by an increase in the number of tokens.

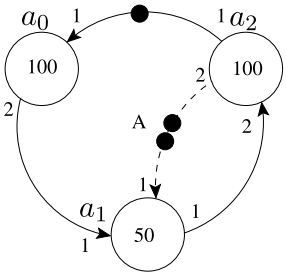


Fig. 2.3 SDF Graph after modeling buffer-size of 2 on the edge from actor a_2 to a_1

Figure 2.3 shows such an example, where the buffer size of the channel from a_1 to a_2 is shown as two. Before a_1 can be executed, it has to check if enough buffer space is available. This is modeled by requiring tokens from the back-edge to be consumed. Since it produces one token per firing, one token from the back-edge is consumed, indicating reservation of one buffer space on the output edge. On the consumption side, when a_2 is executed, it frees two buffer spaces, indicated by a release of two tokens on the back-edge. In the model, the output buffer space is claimed at the start of execution, and the input token space is released only at the end of firing. This ensures atomic execution of the actor.

3 Comparison of Dataflow Models

While SDF graphs allow analysis of many properties and are well-suited for multimedia applications, they do have some restrictions. For example, conditional and data-dependent behaviour cannot be expressed in these models. In this section, we provide an overview of the other models of computation (MoC). In (Stuijk 2007), Stuijk has summarized and compared many models on the basis of their expressiveness and succinctness, efficiency of implementation, and analyzability. Expressiveness determines to what extent real-applications can be represented in a particular model. Models that are static in nature (e.g. SDF) cannot accurately capture behaviour of highly dynamic applications (e.g. object segmentation from an input sequence) accurately. Succinctness (or compactness) determines how compact that representation is. Efficiency of implementation determines how easily the application model can be implemented in terms of its schedule length. Analyzability determines to what extent the model can be analyzed and performance properties of applications determined. As mentioned in the earlier section, this is one of the most important considerations for us. In general, a model that can be easily analyzed at design-time is also more efficient for implementation, since most scheduling and resource assignment decisions can be made at design-time. This implies a lower run-time implementation overhead. Figure 2.4 shows how different models are placed on these three axes.

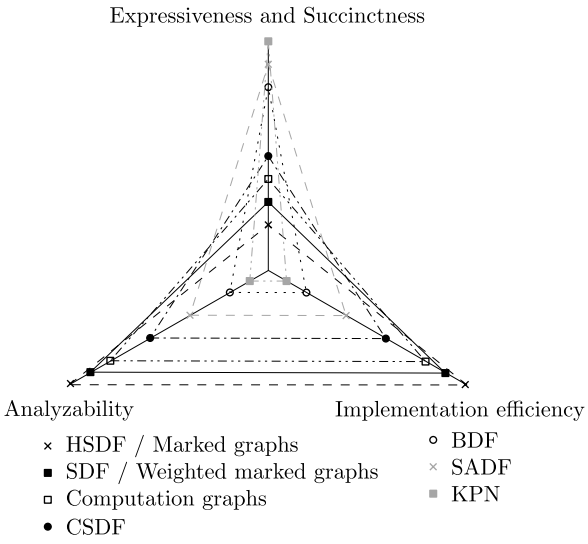


Fig. 2.4 Comparison of different models of computation (Stuijk 2007)

Kahn Process Network

Kahn process network (KPN) was proposed by Kahn in 1974 (Kahn 1974). The amount of data read from an edge may be data-dependent. This allows modeling of any continuous function from the inputs of the KPN to the outputs of the KPN with an arbitrarily small number of processes. KPN is sufficiently expressive to capture precisely all data dependent dataflow transformations. However, this also implies that in order to analyze properties like the throughput or buffer requirements of a KPN all possible input values have to be considered.

Similar to SDF graph, in a KPN, processes communicate via unbounded FIFO channels. While reading from a channel is blocking, i.e. if there is not sufficient data on any of its inputs, the process stalls, writing to an output channel is always non-blocking due to unbounded capacity on the output channels. Processes are not allowed to test an input channel for existence of tokens without consuming them.

Scenario Aware Dataflow

Scenario aware dataflow (SADF) was first introduced by Theelen in 2006 (Theelen et al. 2006). This is also a model for design-time analysis of dynamic streaming and signal processing applications. This model allows for data-dependent behaviour in processes. Each different execution pattern is defined as a *scenario*. Such scenarios denote different modes of operations in which resource requirements can differ considerably. The scenario concept enables to coherently capture the variations in behaviour of different processes in a streaming application. A key novelty of SADF is the use of a stochastic approach to capture the scenario occurrences as well as the

occurrence of different execution times within a scenario in an abstract way. While some properties of these graphs like deadlock and throughput are possible to analyze at design time, in practice this analysis can be quite slow. This model is less compact than KPN, since all scenarios have to be explicitly specified in the model, and known at design time. This also makes it less expressive since not all kinds of systems can be expressed accurately in SADF.

Boolean Dataflow

The last model of computation that we discuss having data-dependent behaviour is boolean dataflow (BDF) model (Lee 1991; Buck and Lee 1993). In this model, each process has a number of inputs and outputs to choose from. Depending on the value of *control tokens* data is read from one of the input channels, and written to one of the output channels. This model is less expressive than the earlier two models discussed, since the control freedom in modeling processes is limited to either true or false. Similar to the earlier two models discussed, the analyzability is limited.

Cyclo Static Dataflow

Now we move on to the class of more deterministic data flow models of computation. In a cyclo-static dataflow (CSDF) model (Lauwereins et al. 1994; Bilsen et al. 1996), the rates of data consumed and produced may change between subsequent firings. However, the pattern of this change is pre-determined and cyclic, making it more analyzable at design time. These graphs may be converted to SDF graphs, and are therefore as expressive as SDF graphs. However, the freedom to change the rates of data makes them more compact than SDF in representing some applications. They are also as analyzable, but slower if we consider the same number of actors, since the resulting schedule is generally a little more complex.

Recently, special channels have been introduced for CSDF graphs (Denolf et al. 2007). Often applications share buffers between multiple consumers. This cannot be directly described in CSDF. The authors show how such implementation specific aspects can still be modeled in CSDF without the need of extensions. Thus, the analyzability of the graph is maintained, and appropriate buffer-sizes can be computed from the application model.

Computation Graphs

Computation graphs were first introduced by Karp and Miller in 1966 (Karp and Miller 1966). In these graphs there is a threshold set for each edge specifying the

minimum number of tokens that should be present on that edge before an actor can fire. However, the number of tokens produced and consumed for each edge is still fixed. These models are less expressive than CSDF, but more analyzable. A synchronous data flow graph is a subset of these computation graphs.

Synchronous Dataflow

Synchronous dataflow (SDF) graphs were first proposed by Lee and Messerschmitt in 1987 (Lee and Messerschmitt 1987). However, as has been earlier claimed (Stuijk 2007), these correspond to subclass weighted marked graph (Teruel et al. 1992) of Petri nets, which is a general purpose model of computation with a number of applications (Petri 1962; Murata 1989). SDF graphs have a constant input and output rate that does not change with input or across different firings. They also don't support execution in different scenarios as may be specified by data. Therefore, their expressivity is rather limited. However, this also makes them a lot easier to analyze. Many performance parameters can be analyzed as explained in Sect. 4.

Homogeneous Synchronous Dataflow

Homogeneous Synchronous dataflow (HSDF) graphs are a subset of SDF graphs. In HSDF graph model, the rates of all input and output edges is one. This implies that only one token is read and written in any firing of an actor. This limits the expressiveness even more, but makes the analysis somewhat easier. HSDF graphs can be converted into SDF and vice-versa. However, in practice the size of an HSDF for an equivalent SDFG may be very large as shown by examples in (Stuijk 2007). Lately, analysis techniques have been developed that work almost as fast directly on an SDF graph as on an HSDF graph (for the same number of nodes) (Ghamarian 2008). Therefore, the added advantage of using an HSDF graph is lost.

After considering all the alternatives, we decided in favour of SDF graphs since their ability to analyze applications in terms of other performance requirements, such as throughput and buffer, was one of our key requirements. Further, a number of analysis tools for SDF graph were available (and more in development) when this research was started (SDF3 2009). Further, a CSDF model of an application can easily be represented in an SDF model. However, since SDF graphs are not able to express dynamism in some real applications accurately, we do have to pay a little overhead in estimating performance. For example, the execution time is assumed to be the worst-case execution-time. Thus, in some cases, the performance estimates may be pessimistic.

4 Performance Modeling

In this section, we define the major terminology that is relevant for this book.

Definition 5 (Actor Execution Time) Actor execution time, $\tau(a)$ is defined as the time needed to complete execution of actor a on a specified node. In cases where the required time is not constant but varying, this indicates the maximum time for actor execution.

$\tau(a_0) = 100$, for example, in Fig. 2.3.

Definition 6 (Iteration) An **iteration** of an SDF graph is defined as the minimum non-zero execution (i.e. at least one actor has executed) such that the state of the graph before that execution is obtained.

In Fig. 2.3, one iteration of graph A is completed when a_0, a_1 and a_2 have each completed one, two and one execution(s) respectively.

Definition 7 (Repetition Vector) Repetition Vector q of an SDF graph A is defined as the vector specifying the number of times each actor in A has to be executed to complete one iteration of A.

For example, in Fig. 2.3, $q[a0\ a1\ a2] = [1\ 2\ 1]$.

Definition 8 (Application Period) Application Period $Per(A)$ is defined as the time SDFG A takes to complete one iteration.

$Per(A) = 300$ in Fig. 2.3, assuming it has sufficient resources and no contention, and all the actors fire as soon as they are ready. (Note that actor a_1 has to execute twice.)

Definition 9 (Application Throughput) Application Throughput, Thr_A is defined as the number of iterations of an SDF graph A in one second.

This is simply the inverse of period, $Per(A)$, when period is defined in seconds. For example, an application with a throughput of 50 Hz takes 20 ms to complete one iteration. When the graph in Fig. 2.3 is executing on a single processor of 300 MHz, the throughput of A is 1 MHz since the period is 1 micro-second.

Throughput is one of the most interesting properties of SDF graphs relevant to the design of any multimedia system. Designers and consumers both want to know the sustained throughput the system can deliver. This parameter often directly relates to the consumer. For example, throughput of an H264 decoder may define how many frames can be decoded per second. A higher throughput in this case directly improves the consumer experience.

Steady-State vs Transient Behaviour

Often some actors of an applications may execute a few times before the periodic behaviour of the application starts. For example, consider the application graph as

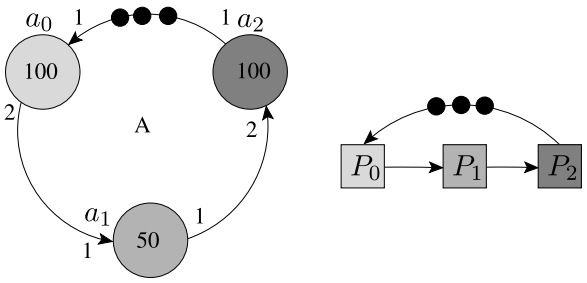


Fig. 2.5 SDF Graph and the multi-processor architecture on which it is mapped

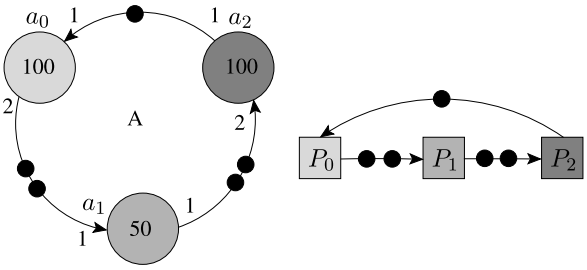


Fig. 2.6 Steady-state is achieved after two executions of a_0 and one of a_1

shown earlier in Fig. 2.1, but now with three initial tokens on the edge from a_2 to a_0. Consider further, that each of the three actors is mapped on a multi-processor system with three processors, P_0, P_1 and P_2, such that actor a_i is mapped on P_i for $i = 0, 1, 2$. Let us assume that the processors are connected to each other with a point-to-point connection with infinite bandwidth with directions similar to channels in the application graph. Figure 2.5 shows the updated graph and the three processor system with the appropriate mapping.

In this example, if we look at the time taken for one single iteration, we get a period of 300 cycles. However, since each actor has its own dedicated processor, soon we get the token distribution as shown in Fig. 2.6. From this point onwards, all the actors can continue firing indefinitely since all actors have sufficient tokens and dedicated resources. Thus, every 100 cycles, an iteration of application A is completed. This final state is called *steady-state*. The initial execution of the graph leading to this state is called *transient phase*.

For the graph as shown in Fig. 2.5, the maximal throughput for 300 MHz processors is 3 MHz or three million iterations per second. In this book when we refer to the throughput of a graph, we generally refer to the maximal achievable throughput of a graph, unless otherwise mentioned. This only refers to the steady-state throughput. When we use the term *achieved throughput* of a graph, we shall refer to the long-term average throughput achieved for a given application. This also includes the transient phase of an application. Please note that for *infinitely long* execution, the long-term average throughput is the same as the steady-state throughput.

Another way to define throughput is the rate of execution of an *output* actor divided by its repetition vector entry. If we consider actor a_2 as the output actor

of application A, we see that the throughput of the application is the same as the execution rate of a_2, since its repetition vector entry is 1.

Throughput Analysis of (H)SDF Graphs

A number of analysis techniques are available to compute the throughput of SDF graphs (Ghamarian 2008; Stuijk 2007; Sriram and Bhattacharyya 2000; Dasdan 2004; Bambha et al. 2002). Many of these techniques first convert an SDF graph into a homogeneous SDF (HSDF) graph. HSDF is a special class of SDF in which the number of tokens consumed and produced is always equal to one. Techniques are available to convert an SDF into HSDF and the other way around (Sriram and Bhattacharyya 2000). After conversion to HSDF, throughput is computed as the inverse of the *maximal cycle mean (MCM)* of the HSDF graph (Dasdan 2004; Karp and Miller 1966). MCM in turn is the maximum of all *cycle-means*. A cycle-mean is computed as the weighted average of total delay in a cycle in the HSDF graph divided by the number of tokens in it.

The conversion to HSDF from an SDF graph may result in an explosion in the number of nodes (Pino and Lee 1995). The number of nodes in the corresponding HSDF graph for an SDF graph is determined by its repetition vector. There are examples of real-applications (H263 in this case), where an SDF model requires only 4 nodes and an HSDF model of the same application has 4754 nodes (Stuijk 2007). This makes the above approach very infeasible for many multimedia applications. Lately, techniques have been presented that operate on SDF graphs directly (Ghamarian et al. 2006; Ghamarian 2008). These techniques essentially simulate the SDF execution and identify when a steady-state is reached by comparing previous states with the current state. Even though the theoretical bound is high, the experimental results show that these techniques can compute the throughput of many multimedia applications within milliseconds.

A tool called SDF^3 has been written and is available on-line for use by the research community (Stuijk et al. 2006a; SDF3 2009). Beside being able to generate random SDF graphs with specified properties, it can also compute throughput of SDF graphs easily. It allows to visualize these graphs as well, and compute other performance properties. The same tool was used for throughput computation and graph generation in many experiments conducted in this book.

However, the above techniques only work for fixed execution times of actors. If there is any change in the actor execution time, the entire analysis has to be repeated. A technique has been proposed in (Ghamarian et al. 2008) that allows variable execution time. This technique computes equations that limit the application period, for a given range of actor execution times. When the exact actor execution time is known, these equations can be evaluated to compute the actual period of the application. This idea is used in Chap. 3 to compute throughput of applications.

It should be mentioned that the techniques mentioned here do not take into account contention for resources like processor and memory, and essentially assume that infinite resources are available, except SDF^3. SDF^3 also takes resource

contention into account but is limited to preemptive systems. Before we see how throughput can be computed when considering limited computation resources, we review the basic techniques used for scheduling dataflow graphs.

5 Scheduling Techniques for Dataflow Graphs

One of the key aspects in designing multiprocessor systems from any MoC is scheduling. Scheduling, as defined earlier, is the process of determining when and where tasks are executed. Compile-time scheduling promises near-optimal performance at low cost for final system, but is only suitable for static applications. Run-time scheduling can address a wider variety of applications, at greater system cost. Scheduling techniques can be classified in a number of categories based on which decisions are made at compile time (also known as design-time) and which decisions are made at run-time (Lee and Ha 1989; Sriram and Bhattacharyya 2000). There are three main decisions when scheduling tasks (or actors) on a processor: (1) which tasks to **assign** to a given processor, (2) what is the **order** of these tasks on it and (3) what is the precise**timing** of these tasks. (When considering mapping of channels on the network, there are many more categories.) We consider four different types of schedulers.

(1) The first one is **fully static**, where everything is decided at compile time and the processor has to simply execute the tasks. This approach has traditionally been used extensively for DSP applications due to their repetitive and constant resource requirement. This is also good for systems where guarantees are more important and (potential) speed-up from earlier execution is not desired. Further, the run-time scheduler becomes very simple since it does not need to check for availability of data and simply executes the scheduled actors at respective times. However, this mechanism is completely static, and cannot handle any dynamism in the system like run-time addition of applications, or any unexpectedly higher execution time for a particular iteration.

(2) The second type is **self-timed**, where the assignment and ordering is already done at compile time. However, the exact time for firing of actors is determined at run-time, depending on the availability of data. Self-timed scheduling is more suitable for cases when the execution time of tasks may be data-dependent, but the variation is not very large. This can often result in speed-up of applications as compared to analysis at design-time, provided the worst case execution time estimates are used for analyzing the application performance. Since earlier arrival of data cannot result in later production of data – a property defined as *monotonicity*, the performance bounds computed at compile-time are preserved. However, this also implies that the schedule may become non-work-conserving, i.e. that a task may be waiting on a processor, while the processor is sitting idle waiting for the task in order.

(3) The third type is **static assignment**, where the mapping is already fixed at compile time, but the ordering and timing is done at run-time by the scheduler. This

Table 2.1 The time which the scheduling activities "assignment", "ordering", and "timing" are performed is shown for four classes of schedulers. The scheduling activities are listed on top and the strategies on the left (Lee and Ha 1989)

Scheduler	Assignment	Ordering	Timing	Work-conserving
Fully static	compile	compile	compile	no
Self timed	compile	compile	run	no
Static assignment	compile	run	run	processor-level
Fully dynamic	run	run	run	yes

allows the schedule to become work-conserving and perhaps achieve a higher overall throughput in the system. However, it might also result in a lower overall throughput since the bounds cannot be computed at compile-time (or are not preserved in this scheduler). This scheduling is most applicable for systems where applications have a large variation in execution time. While for a single application, the order is still imposed by the data-dependency among tasks and makes self-timed more suitable, for multiple applications the high variation in execution time, makes it infeasible to enforce the static-order.

(4) The last one is called **fully dynamic**, where mapping, ordering and timing are all done at run-time. This gives full freedom to the scheduler, and the tasks are assigned to an idle processor as soon as they are ready. This scheduler also allows for task migration. This may result in a yet higher throughput, since this tries to maximize the resource utilization and minimize the idle time, albeit at the cost of performance guarantee. It should be noted that run-time assignment also involves a (potentially) higher overhead in data movement. When the assignment is fixed at compile time, the task knows the processor to which the receiving actor is mapped *a priori*.

These four scheduling mechanisms are summarized in Table 2.1. As we move from fully-static to fully-dynamic scheduler, the run-time scheduling activity (and correspondingly overhead) increases. However, this also makes it more robust for handling dynamism in the system. The last column shows the work-conserving nature of the schedulers. A fully-static scheduler is non-conserving since the exact time and order of firing (task execution) is fixed. The self-timed schedule is work-conserving only when at most one task is mapped on one processor, while the static-assignment is also work-conserving for multiple applications. However, in static assignment if we consider the whole system (i.e. multiple processors), it may not be work-conserving, since tasks may be waiting to execute on a particular processor, while other processors may be idle.

In a homogeneous system, there is naturally more freedom to choose which task to assign to a particular processor instance, since all processors are identical. On the contrary, in a heterogeneous system this freedom is limited by which processors can be used for executing a particular task. When only one processor is available for a particular task, the mapping is inherently dictated by this limitation. For a complete heterogeneous platform, a scheduler is generally not fully dynamic, unless a task is allowed to be mapped on different types of processors. However, even

in those cases, assignment is usually fixed (or chosen) at compile time. Further, the execution time for multimedia applications is generally highly variant making a fully-static scheduler often infeasible. Designers therefore have to make a choice between a self-timed or a static-assignment schedule. The only choice left is essentially regarding the ordering of tasks on a processor.

In the next section, we shall see how to analyze performance of multiple applications executing on a multiprocessor platform for both self-timed and static-assignment scheduler. Since the only difference in these two schedulers is the time at which ordering of actors is done, we shall refer to self-timed and static-assignment scheduler as **static-ordering** and **dynamic-ordering** scheduler respectively for easy differentiation.

6 Analyzing Application Performance on Hardware

In Sect. 4, we assumed we have infinite computing and communication resources. Clearly, this is not a valid assumption. Often processors are shared, not just among tasks of one application, but also with other applications. We first see how we can model this resource contention for a single application, and later for multiple applications.

We start with considering an HSDF graph with constant execution times to illustrate that even for HSDF graphs it is already complicated. In (Bambha et al. 2002), the authors propose to analyze performance of a *single application* modeled as an HSDF graph mapped on a multi-processor system by modeling dependencies of resources by adding extra edges to the graph. Adding these extra edges enforces a strict order among the actors mapped on the same processor. Since the processor dependency is now modeled in the graph itself, we can simply compute the maximum throughput possible of the graph, and that corresponds to the maximum performance the application can achieve on the multiprocessor platform.

Unfortunately, this approach does not scale well when we move on to the SDF model of an application. Converting an SDF model to an HSDF model can potentially result in a large number of actors in the corresponding HSDF graph. Further, adding such resource dependency edges essentially enforces a static-order among actors mapped on a particular processor. While in some cases, only one order is possible (due to natural data dependency among those actors), in other cases the number of different orders is also very high. Further, different orders may result in different overall throughput of the application. The situation becomes even worse when we consider multiple applications. This is shown by means of an example in the following sub-section.

Static Order Analysis

In this section, we look at how application performance can be computed using static-order scheduler, where both processor assignment and ordering is done at

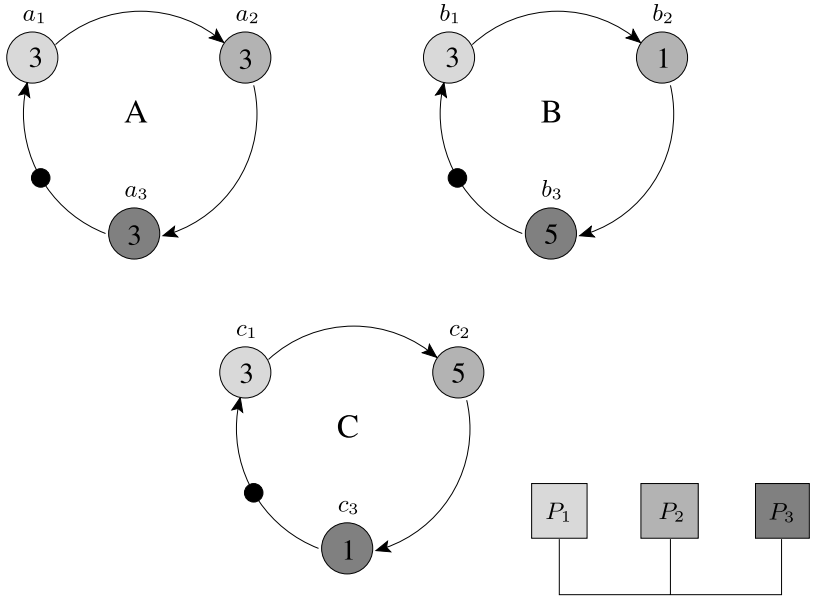


Fig. 2.7 Example of a system with 3 different applications mapped on a 3-processor platform

compile-time. We show that when multiple applications are mapped on multiple processors sharing them, it is (1) difficult to make a static schedule, (2) time-consuming to analyze application performance given a schedule, and (3) infeasible to explore the entire scheduling space to find one that gives the best performance for all applications.

Three application graphs – A, B and C, are shown in Fig. 2.7. Each is an HSDF with three actors. Let us assume actors T_i are mapped on processing node P_i where T_i refers to a_i, b_i and c_i for $i = 1, 2, 3$. This contention for resources is shown by the dotted arrows in Fig. 2.8. Clearly, by putting these dotted arrows, we have fixed the actor-order for each processor node. Figure 2.8 shows just one such possibility when the dotted arrows are used to combine the three task graphs. Extra tokens have been inserted in these dotted edges to indicate the initial state of arbiters on each processor. The tokens indicating processor contention are shown in grey, while the regular data tokens are shown in black. Clearly, this is only possible if each task is required to be run an equal number of times. If the rates of each task are not the same, we need to introduce multiple copies of actors to achieve the required ratio, thereby increasing analysis complexity.

When throughput analysis is done for this complete graph, we obtain a mean cycle count of 11. The bold arrows represent the edges that limit the throughput. The corresponding schedule is also shown. One actor of each application is ready to fire at instant t_0. However, only a_1 can execute since it is the only one with a token on all its incoming edges. We find that the graph soon settles into the periodic schedule of 11 clock cycles. This period is denoted in the schedule diagram of Fig. 2.8 between the time instant t_1 and t_2.

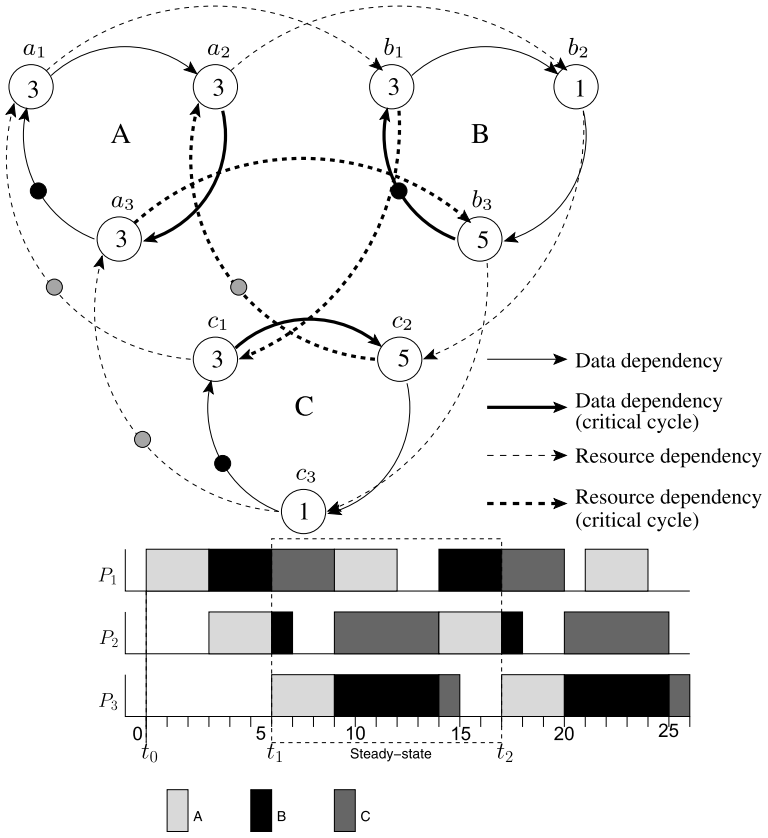


Fig. 2.8 Graph with clockwise schedule (static) gives MCM of 11 cycles. The critical cycle is shown in bold

Figure 2.9 shows just another of the many possibilities for ordering the actors of the complete HSDF (we reversed the ordering of resources between the actors). Interestingly, the mean cycle count for this graph is 10, as indicated by the bold arrows. In this case, the schedule starts repeating after time t_1, and the steady state length is 20 clock cycles, as indicated by the difference in time instants t_1 and t_2. However, since two iterations for each application are completed, the average period is only 10 clock cycles.

From arbitration point of view, if application graphs are analyzed in isolation, there seems to be no reason to prefer actor b_1 or c_1 after a_1 has finished executing on P_1. There is at least a delay of 6 clock cycles before a_1 needs P_1 again. Also, since b_1 and c_1 take only 3 clock cycles each, 6 clock cycles are enough to finish their execution. Further both are ready to be fired, and will not cause any delay. Thus, the local information about an application and the actors that need a processor resource does not easily dictate preference of one task over another. However, as we see in this example, executing c_1 is indeed better for the overall performance.

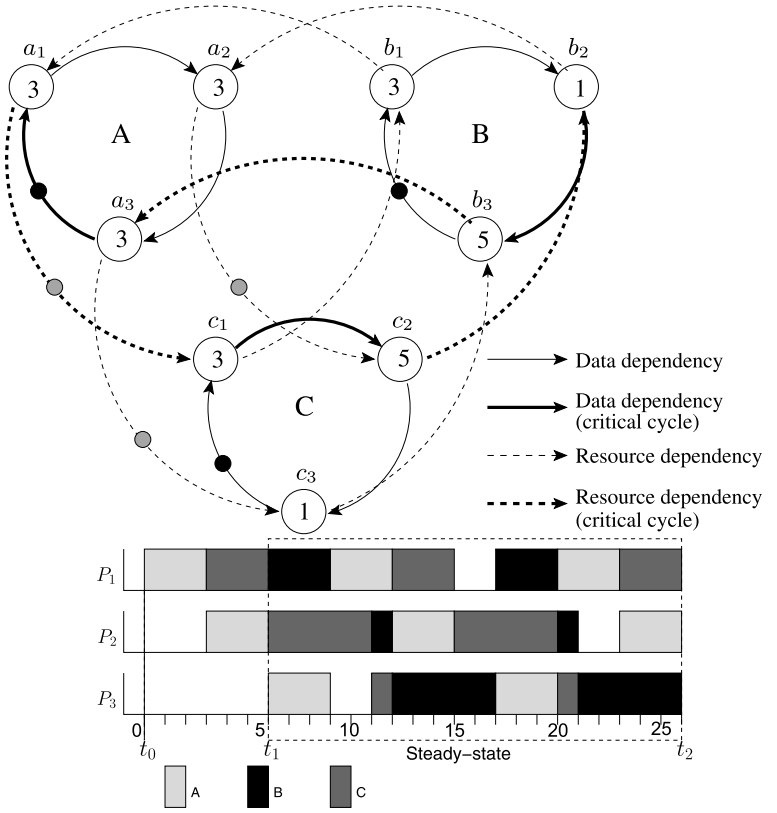


Fig. 2.9 Graph with anti-clockwise schedule (static) gives MCM of 10 cycles. The critical cycle is shown in bold. Here two iterations are carried out in one steady-state iteration

Computing a static order relies on the global information and produces the optimal performance. This becomes a serious problem when considering MPSoC platforms, since constructing the overall HSDF graph and then computing its throughput is very compute intensive. Further, this is not suitable for dynamic applications. A small change in execution time may change the optimal schedule.

The number of possibilities for constructing the HSDF from individual graphs is very large. In fact, if one tries to combine g graphs of say a actors, scheduled in total on a processors, there are $((g-1)!)^a$ unique combinations, each with a different actor ordering. (Each processor has g actors to schedule, and therefore $(g-1)!$ unique orderings on a single processor. This leads to $((g-1)!)^a$ unique combinations, since ordering on each processor is independent of ordering on another.) To get an idea of vastness of this number, if there are 5 graphs with 10 actors each we get 24^{10} or close to $6.34 \cdot 10^{13}$ possible combinations. If each computation would take only 1 ms to compute, 2009 years are needed to evaluate all possibilities. This is only considering the cases with equal rates for each application, and only for HSDF graphs. A typical SDF graph with different execution rates would only make the

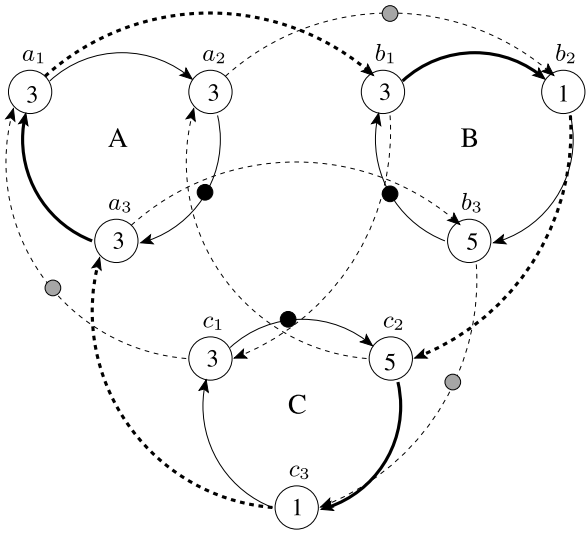


Fig. 2.10 Deadlock situation when a new job, C arrives in the system. A cycle a_1, b_1, b_2, c_2, c_3, a_3, a_1 is created without any token in it

problem even more infeasible, since the transformation to HSDF may yield many actor copies. An exhaustive search through all the graphs to compute optimal static order is simply not feasible.

We can therefore conclude that computing a static order for multiple applications is very compute intensive and infeasible. Further, the performance we obtain may not be optimal. However, the advantage of this approach is that we are guaranteed to achieve the performance that is analyzed for *any* static order at design-time *provided the worst-case execution time estimates are correct*.

Deadlock Analysis

Deadlock avoidance and detection is an important concern when applications may be activated dynamically. Applications modeled as (H)SDF graphs can be analyzed for deadlock occurrence within each (single) application. However, deadlock detection and avoidance between multiple applications is not so easy. When static order is being used, every new use-case requires a new schedule to be loaded into the platform at run-time. A naive reconfiguration strategy can easily send the system into deadlock. This is demonstrated with an example in Fig. 2.10.

Say actors a_2 and b_3 are running in the system on P_2 and P_3 respectively. Further assume that a static order for each processor currently is $A \rightarrow B$ when only these two applications are active, and with a third application C, $A \rightarrow B \rightarrow C$ for each node. When application C is activated, it gets P_1 since it is idle. Let us see what happens to P_2: a_2 is executing on it and it is then assigned to b_2. P_3 is assigned to c_3 after b_3 is done. Thus, after each actor is finished executing on its currently assigned processor, we get a_3 waiting for P_3 that is assigned to task c_3, b_1 waiting for P_1 which is assigned to a_1, and c_2 waiting for P_2, which is assigned to b_2.

Looking at Fig. 2.10, it is easy to understand why the system goes into a deadlock. The figure shows the state when each actor is waiting for a resource and not able to execute. The tokens in the individual sub-graph show which actor is ready to fire, and the token on the dotted edge represents which resource is available to the application. In order for an actor to fire, a token should be present on all its incoming edges – in this case both on the incoming dotted edge and the solid edge. It can be further noted that a cycle is formed without any token in it. This is clearly a situation of deadlock (Karp and Miller 1966) since the actors on this cycle will never be enabled. This cycle is drawn in Fig. 2.10 with bold edges. It is possible to take special measures to check and prevent the system from going into such a deadlock. This, however, implies extra overhead at both compile-time and run-time. The application may also have to be delayed before it can be admitted into the system.

Dynamic Order Analysis

In this section, we look at static-assignment scheduling, where only processor assignment is done at compile-time and the ordering is done at run-time. First-come-first-serve (FCFS) falls under this category. Another arbiter that we propose here in this category is round-robin-with-skipping (RRWS). In RRWS, a recommended order is specified, but the actors can be skipped over if they are not ready when the processor becomes idle. This is similar to the fairness arbiter proposed by Gao in 1983 (Gao 1983). However, in that scheduler, all actors have equal weight. In RRWS, multiple instances of an actor can be scheduled in one cycle to provide an easy rate control mechanism.

The price a system-designer has to pay when using dynamic scheduling is the difficulty in determining application performance. Analyzing application performance when multiple applications are sharing a multiprocessor platform is not easy. An approach that models resource contention by computing **worst case response time** for TDMA scheduling (requires preemption) has been analyzed in (Bekooij et al. 2005). This analysis also requires limited information from the other SDFGs, but gives a very conservative bound that may be too pessimistic. As the number of applications increases, the minimum performance bound decreases much more than the average case performance. Further, this approach assumes a preemptive system. A similar worst-case analysis approach for round-robin is presented in (Hoes 2004), which also works for non-preemptive systems, but suffers from the same problem of lack of scalability.

Let us revisit the example in Fig. 2.7. Since 3 actors are mapped on each processor, an actor may need to wait when it is ready to be executed at a processor. The maximum waiting time for a particular actor can be computed by considering the *critical instant* as defined by Liu and Layland (Liu and Layland 1973). The **critical instant** for an actor is defined as an instant at which a request for that actor has the largest response time. The **response time** is defined as the sum of an actor's waiting time and its execution time. If we take worst case response time, this can

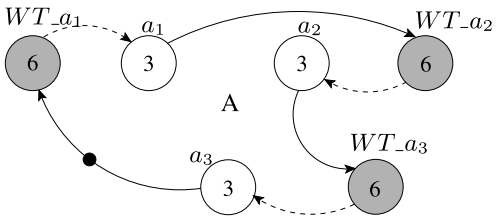


Fig. 2.11 Modeling worst case waiting time for application *A* in Fig. 2.7

Table 2.2 Table showing the deadlock condition in Fig. 2.10

Node	Assigned to	Task waiting and reassigned in RRWS
P1	A	B
P2	B	C
P3	C	A

be translated as the instant at which we have the largest waiting time. For dynamic scheduling mechanisms, it occurs when an actor becomes ready just after all the other actors, and therefore has to wait for all the other actors. Thus, the total waiting time is equal to the sum of worst-case execution times of all the other actors on that particular node and given by the following equation.

$$t_{wait}(T_{ij}) = \sum_{k=1,\, k\neq i}^{m} t_{exec}(T_{kj}) \tag{2.1}$$

Here $t_{exec}(T_{ij})$ denotes the worst case execution time of actor T_{ij}, i.e. actor of task T_i mapped on processor j. This leads to a waiting time of 6 time units as shown in Fig. 2.11. An extra node has been added for each 'real' node to depict the waiting time (WT_a_i). This suggests that each application will take 27 time units in the worst case to finish execution. This is the maximum period that can be obtained for applications in the system, and is therefore guaranteed. However, as we have seen in the earlier analysis, the applications will probably settle for a period of 10 or 11 cycles depending on the arbitration decisions made by the scheduler. Thus, the bound provided by this analysis is about two to three times higher than real performance.

The deadlock situation shown in Fig. 2.10 can be avoided quite easily by using dynamic-order scheduling. Clearly, for FCFS, it is not an issue since resources are never blocked for non-ready actors. For RRWS, when the system enters into a deadlock, the arbiter would simply skip to the actor that is ready to execute. Thus, processors 1, 2 and 3 are reassigned to B, C and A as shown in Table 2.2. Further, an application can be activated at any point in time without worrying about deadlock. In dynamic scheduling, there can never be a deadlock due to dependency on processing resources for atomic non-preemptive systems.

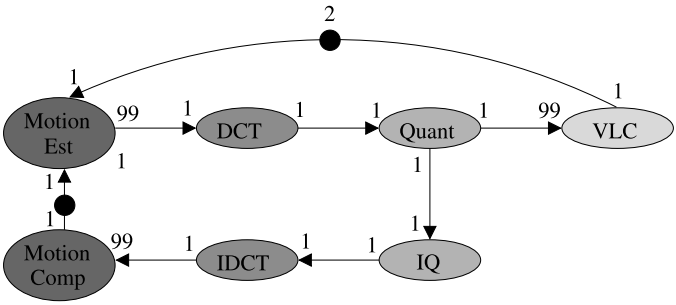
7 Composability

As highlighted in Chap. 1, one of the key challenges when designing multimedia systems is dealing with multiple applications. For example, a mobile phone supports various applications that can be active at the same time, such as listening to mp3 music, typing an sms and downloading some java application in the background. Evaluating resource requirements for each of these cases can be quite a challenge even at design time, let alone at run time. When designing a system, it is quite useful to be able to estimate resource requirements early in the design phase. Design managers often have to negotiate with the product divisions for the overall resources needed for the system. These estimates are mostly on a higher level, and the managers usually like to adopt a *spread-sheet* approach for computing it. As we see in this section, it is often not possible to use this view.

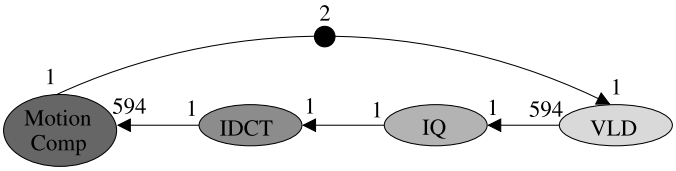
We define *composability* as mapping and analysis of performance of multiple applications on a multiprocessor platform in isolation, i.e. requiring limited information from other applications. Note that this is different from what has been defined in literature by Kopetz (Kopetz and Obermaisser 2002; Kopetz and Suri 2003). Composability as defined by Kopetz is *integration of a whole system from well-specified and pre-tested sub-systems without unintended side-effects*. The key difference between this definition and our definition is that composability as defined by Kopetz is a property of a system such that the performance of applications in isolation and running concurrently with other applications is the same. For example, say we have a system with 10 applications, each with only one task and all mapped on the same processor. Let us further assume that all tasks take 100 time units to execute in isolation. According to the definition of Kopetz, it will also take 100 time units when running with the other tasks. This can only be achieved in two ways.

(1) The platform supports complete *virtualization* of resources, and each application gets one-tenth of processor resources. This implies that we only use one-tenth of the resources even when only one application is active. Further, to achieve complete virtualization, the processor has to be preempted and its context has to be switched every single cycle. (This could be relaxed a bit, depending on the observability of the system.)
(2) We consider a worst-case schedule in which all applications are scheduled, and the total execution time of all applications is 100 time units. Thus, if a particular application is not active, the processor simply waits for that many time units as it is scheduled for. This again leads to under-utilization of the processor resources. Besides, if any application takes more time, then the system may collapse.

Clearly, this implies that we cannot harness the full processing capability. In a typical system, we would want to use this compute power to deliver a better quality-of-service for an application when possible. We want to let the system execute as many applications as possible with the current resource availability, and let applications achieve their best behaviour possible in the given use-case. Thus, in the example with 10 applications, if each application can run in 10 time units in isolation, it might take 100 time units when running concurrently with all the other applications.



(a) SDF model of H263 encoder



(b) SDF model of H263 decoder

Fig. 2.12 SDF graphs of H263 encoder and decoder

We would like to predict the application properties given the application mix of the system, with as little information from other applications as possible.

Some of the things we would like to analyze are for example, deadlock occurrence, and application performance. Clearly, since there is more than one application mapped on a multi-processor system, there will be contention for the resources. Due to this contention, the throughput analyzed for an application in isolation is not always achievable when the application runs together with other applications. We see how different levels of information from other applications affect analysis results in the next sub-section.

Performance Estimation

Let us consider a scenario of video-conferencing in a hand-held device. Figure 2.12 shows SDF graphs for both H263 encoding and decoding applications. The encoder model is based on the SDF graph presented in (Oh and Ha 2004), and the decoder model is based on (Stuijk 2007). (The self-edges are removed for simplicity.) The video-stream assumed for the example is of QCIF resolution that has 99 macro-blocks to process, as indicated by the rates on the edges. Both encoding and decoding have an actor that works on variable length (VLC and VLD respectively), quantization (Quant and IQ respectively), and discrete cosine transform (DCT and IDCT respectively). Since we are considering a heterogeneous system, the processor responsible for an actor in encoding process is usually responsible for the corresponding decoding actor. E.g. when the encoding and decoding are done concurrently, the

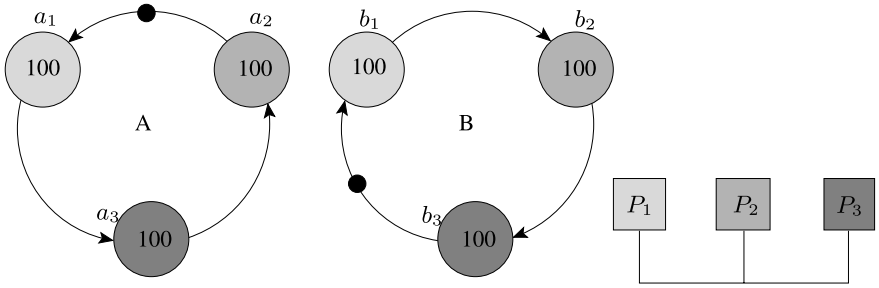


Fig. 2.13 Two applications running on same platform and sharing resources

DCT and IDCT are likely to be executed on the same processor, since that processor is probably more suited for cosine transforms. This resource dependency in the encoder and decoder models is shown by shading in Fig. 2.12. Since decoding works in opposite order as compared to encoding, the resource dependency in encoding and decoding is exactly reversed. A similar situation happens during decoding and encoding of an audio stream as well.

The above resource dependencies may cause surprising performance results, as shown by the example in Fig. 2.13. The figure shows an example of two application graphs A and B with three actors each, mapped on a 3-processor system. Actors a_1 and b_1 are mapped on p_1, a_2 and b_2 are mapped on p_2, and a_3 and b_3 are mapped on p_3. Each actor takes 100 clock cycles to execute. While both applications A and B might look similar, the dependency in A is anti-clockwise and in B clockwise to highlight the situation in the above example of simultaneous H263 encoding and decoding.

Let us try to *add* the resource requirement of actors and applications, and try to reason about their behaviour when they are executing concurrently. Each processor has two actors mapped; each actor requires 100 time units. If we limit the information to only actor-level, we can conclude that one iteration of each a_1 and b_1 can be done in a total of 200 time units on processor P_1, and the same holds for processors P_2 and P_3. Thus, if we consider a total of 3 million time units, each application should finish 15,000 iterations, leading to 30,000 iterations in total. If we now consider the graph-level local information only, then we quickly realize that since there is only one initial token, the minimum period of the applications is 300. Thus, each application can finish 10,000 iterations in 3 million time units. As it turns out, none of these two estimates are achievable.

Let us now increase the information we use to analyze the application performance. We consider the worst-case response time as defined in Eq. 2.1 for each actor. This gives us an upper bound of 200 time units for each actor. If we now use this to compute our application period, we obtain 600 time units for each application. This translates to 5,000 iterations per application in 3 million time units. This is the guaranteed lower bound of performance. If we go one stage further, and try to analyze the full schedule of this two-application system by making a static schedule, we obtain a schedule with a steady-state of 400 time units in which each application

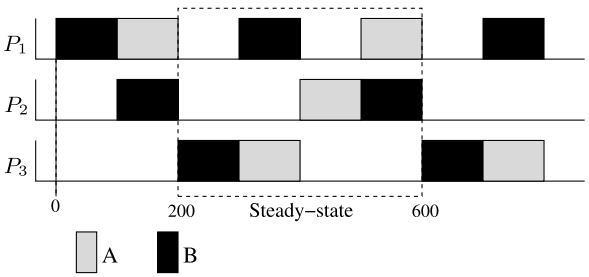


Fig. 2.14 Static-order schedule of applications in Fig. 2.13 executing concurrently

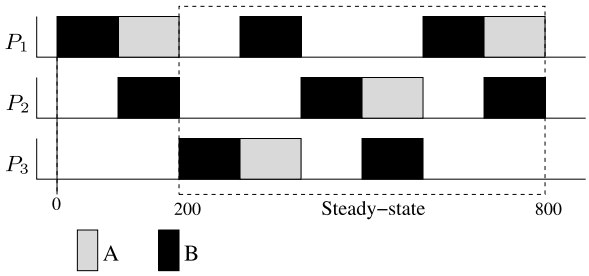


Fig. 2.15 Schedule of applications in Fig. 2.13 executing concurrently when B has priority

completes one iteration. The corresponding schedule is shown in Fig. 2.14. Unlike the earlier predictions, this performance is indeed what the applications achieve. They will both complete one iteration every 400 time units. If we consider dynamic ordering and let the applications run on their own, we might obtain the same order as in Fig. 2.14, or we might get the order as shown in Fig. 2.15. When the exact execution order is not specified, depending on the scheduling policy the performance may vary. If we consider a first-come-first-serve approach, it is hard to predict the exact performance since the actors have equal execution time and they arrive at the same time. If we assume for some reason, application A is checked first, then application A will execute twice as often as B, and vice-versa. The schedule in Fig. 2.15 assumes application B has preference when both are ready at the exact same time. The same behaviour is obtained if we consider round-robin approach with skipping. Interestingly, the number of combined application iterations are still 15,000 – the same as when static order is used.

Table 2.3 shows how different estimating strategies can lead to different results. Some of the methods give a false indication of processing power, and are not achievable. For example, in the second column only the actor execution time is considered. This is a very naive approach and would be the easiest to estimate. It assumes that all the processing power that is available for each node is shared between the two actors equally. As we vary the information that is used to make the prediction, the performance prediction also varies. This example shows why composability needs to be examined. Individually each application takes 300 time units to complete an iteration and requires only a third of the available processor resources. However, when another application enters in the system, it is not possible to schedule both of them with their lowest period of 300 time units, even though the total request for a node is only two-third. Even when preemption is considered, only one application

Table 2.3 Estimating performance: iteration-count for each application in 3,000,000 time units

Appl.	Only actors	Only graph	WC analysis (both graphs)	Static	RRWS/FCFS	
					A pref	B pref
A	15,000	10,000	5,000	7,500	10,000	5,000
B	15,000	10,000	5,000	7,500	5,000	10,000
Total	30,000	20,000	10,000	15,000	15,000	15,000
Proc Util	1.00	0.67	0.33	0.50	0.50	0.50

Table 2.4 Properties of scheduling strategies

Property	Static order	Dynamic order
Design time overhead		
Calculating Schedules	−−	++
Run-time overhead		
Memory requirement	−	++
Scheduling overhead	++	+
Predictability		
Throughput	++	−−
Resource Utilization	+	−
New job admission		
Admission criteria	++	−−
Deadlock-free guarantee	−	++
Reconfiguration overhead	−	+
Dynamism		
Variable Execution time	−	+
Handling new use-case	−−	++

can achieve the period of 300 time units while the other of 600. The performance of the two applications in this case corresponds to the last two columns in Table 2.3. Thus, predicting application performance when executing concurrently with other applications is not very easy.

8 Static vs Dynamic Ordering

Table 2.4 shows a summary of various performance parameters that we have considered, and how static-order and dynamic-order scheduling strategy performs considering these performance parameters. The static-order scheduling clearly has a higher design-time overhead of computing the static order for each use-case. The run-time scheduler needed for both static-order and dynamic-order schedulers is quite simple, since only a simple check is needed to see when the actor is active and ready to

fire. The memory requirement for static scheduling is however, higher than that for a dynamic mechanism. As the number of applications increases, the total number of potential use-cases rises exponentially. For a system with 10 applications in which up to 4 can be active at the same time, there are approximately 400 possible combinations – and it grows exponentially as we increase the number of concurrently active applications. If static ordering is used, besides computing the schedule for all the use-cases at compile-time, one also has to be aware that they need to be stored at run-time. The scalability of using static scheduling for multiple applications is therefore limited.

Dynamic ordering is more scalable in this context. Clearly in FCFS, there is no such overhead as no schedule is computed beforehand. In RRWS, the easiest approach would be to store all actors for a processor in a schedule; when an application is not active, its actors are simply skipped, without causing any trouble for the scheduler. It should also be mentioned here that if an actor is required to be executed multiple times, one can simply add more copies of that actor in this list. In this way, RRWS can provide an easy rate-control mechanism.

The static order approach certainly scores better than a dynamic one when it comes to predictability of throughput and resource utilization. Static-order approach is also better when it comes to admitting a new application in the system since the resource requirements prior and after admitting the application are known at design time. Therefore, a decision whether to accept it or not is easier to make. However, extra measures are needed to reconfigure the system properly so that the system does not go into deadlock as mentioned earlier.

A dynamic approach is able to handle dynamism better than static order since orders are computed based on the worst-case execution time. When the execution-time varies significantly, a static order is not able to benefit from early termination of a process. The biggest disadvantage of static order, however, lies in the fact that any change in the design, e.g. adding a use-case to the system or a new application, cannot be accommodated at run-time. The dynamic ordering is, therefore, more suitable for designing multimedia systems. In the following chapter, we show techniques to predict performance of multiple applications executing concurrently.

9 Conclusions

In this chapter, we began with motivating the need of having an application model. We discussed several models of computation that are available and generally used. Given our application requirements and strengths of the models, we chose the synchronous dataflow (SDF) graphs to model application. We provided a short introduction to SDF graphs and explained some important concepts relevant for this book, namely modeling auto-concurrency and modeling buffer-sizes on channels. We explained how various performance characteristics of an SDF graph can be derived.

The scheduling techniques used for dataflow analysis were discussed and classified depending on which of the three things – assignment, ordering, and timing – are done at compile-time and which at run-time. We highlighted two arbiter classes –

static and dynamic ordering, which are more commonly used, and discussed how application performance can be analyzed considering hardware constraints for each of these arbiters.

We then highlighted the issue of *composability* – mapping and analysis of performance of multiple applications on a multiprocessor platform in isolation. We demonstrated with a small, but realistic example, how predicting performance can be difficult when even small applications are considered. We also saw how arbitration plays a significant role in determining the application performance. We summarized the properties that are important for an arbiter in a multimedia system, and decided that considering the high dynamism in multimedia applications, the dynamic-ordering is more suitable.

Chapter 3
Probabilistic Performance Prediction

As mentioned earlier in Chap. 1, in modern multimedia systems, multiple applications execute concurrently. While traditionally a mobile phone had to support only a handful of applications like communicating with the base station, sending and receiving short messages, and encoding and decoding voice; modern high-end mobile devices also act as a music and video player, camera, gps, mobile TV and a complete personal digital assistant. Due to a huge number of possible combinations of these multiple applications, it becomes a challenge to predict their performance in advance. One of the key design automation challenges are designing systems for these use-cases and fast exploration of software and hardware implementation alternatives with accurate performance evaluation of these use-cases. The number of use-cases are already exponential. When we consider the possibilities of mapping application actors on processors and other resources, the total number of design points that need to be evaluated becomes even larger. A quick but accurate performance analysis technique is therefore very important.

This becomes even more important when applications may be dynamically started and stopped in the system. *Mis-prediction* may result in reduced quality of applications and lower the user-experience. To further complicate matters, the user also expects to be able to download applications at run-time that may be completely unknown to the system designer, for example, a security application running in the background to protect the mobile phone against theft. While some of these applications may not be so critical for the user-experience (e.g. browsing a web), others like playing video and audio are some functions where a reduced performance is easily noticed. Accurate performance prediction is therefore essential to be performed at run-time before starting a new application, and not always feasible at design-time.

While this analysis is well understood (and relatively easier) for preemptive systems (Liu and Layland 1973; Davari and Dhall 1986; Baruah et al. 1996), non-preemptive scheduling has received considerably less attention. However, for high-performance embedded systems (like cell-processing engine (SPE) (Kahle et al. 2005) and graphics processors), non-preemptive systems are preferred over preemptive scheduling for a number of reasons (Jeffay et al. 1991). In many practical systems, properties of device hardware and software either make the preemption impossible or prohibitively expensive. Further, non-preemptive scheduling algorithms

A. Kumar et al., *Multimedia Multiprocessor Systems*, Embedded Systems,
DOI 10.1007/978-94-007-0083-3_3,

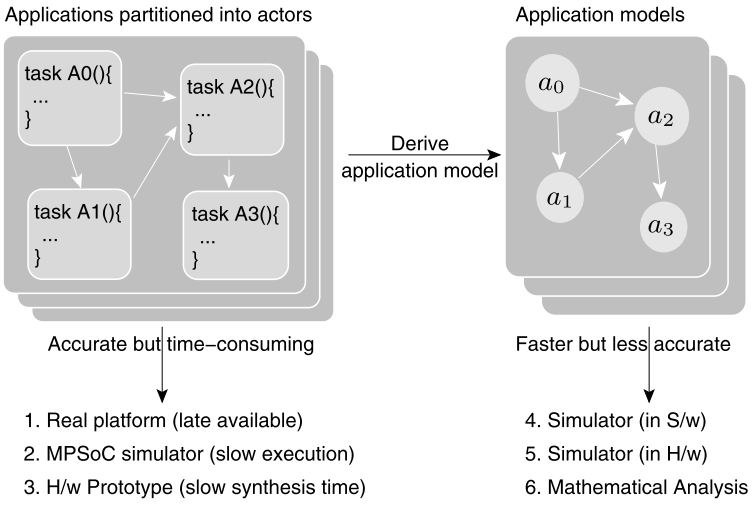


Fig. 3.1 Comparison of various techniques for performance evaluation

are easier to implement than preemptive algorithms and have dramatically lower overhead at run-time (Jeffay et al. 1991). Further, even in multi-processor systems with preemptive processors, some processors (or coprocessors/accelerators) are usually non-preemptive; for such processors, non-preemptive analysis is still needed. It is therefore important to investigate non-preemptive multi-processor systems.

Figure 3.1 puts different approaches for performance evaluation in perspective. The way to obtain most realistic performance estimates is measuring it on the real system. However, this is often not available till late in the design process. An alternative is simulating the (partitioned) application code on a multiprocessor simulation platform that models all the details, like an MPARM simulator. However, this is rather slow. System hardware prototypes on an FPGA are also a viable alternative that is faster once the platform is available. However, this often implies a high synthesis time making the approach infeasible for design space exploration (DSE). In order to reduce this time, application models may be derived that simulate the behaviour of applications on a high level. These models may then be simulated using a transaction level simulator that also takes the architecture and mapping into account. Besides software, some hardware platforms are also available for this simulation (Wawrzynek et al. 2007). The benefit of using such a simulator is that it is much faster than a cycle-accurate simulator or synthesizing a prototype for FPGA. However, when dealing with large number of use-cases, this approach may still not be feasible for DSE, and certainly not for run-time implementation. To further speed performance estimation, analysing models mathematically is the best.

When applications are modeled as synchronous dataflow (SDF) graphs, their performance on a (multi-processor) system can be easily computed when they are executing *in isolation* (provided we have a *good* model). When they execute concurrently, depending on whether the used scheduler is static or dynamic, the arbitration

on a resource is either fixed at design-time or chosen at run-time respectively (as we have seen in Chap. 2). In the former case, the execution order can be modeled in the graph, and the performance of the entire use-case can be determined. The contention is therefore modeled as dependency edges in the SDF graph. An example of such model is presented in Fig. 2.8. However, this is more suited for static applications. For dynamic applications such as multimedia, a dynamic scheduler is more suitable, as has already been motivated in the previous chapter. A static scheduler is not able to deal with varying execution times and actor execution rates; something that is quite typical of dynamic applications. For dynamic scheduling approaches, the contention can be modeled as waiting time for a task, which is added to the execution time to give the total response time. The performance can be determined by computing the performance (throughput) of this resulting SDF graph. With lack of good techniques for accurately predicting the time spent in contention, designers have to resort to worst-case waiting time estimates, that lead to over-designing the system. An example of this is shown in Fig. 2.11, where the waiting time of each actor is equal to the sum of worst-case execution time of other actors. Further, those approaches are not scalable and the over-estimate increases with the number of applications.

In this chapter, a novel probabilistic performance prediction (P^3) algorithm is presented for predicting performance of multiple applications executing concurrently on multi-processor platforms. The algorithm predicts the time that tasks have to spend during the contention phase for a resource. Each application contains a number of *tasks* that have a worst-case execution time. Two approaches are presented – basic and iterative probabilistic techniques. The basic P^3 approach looks at all the possible combinations of actors blocking a particular actor. Since the number of combinations is exponential in the number of actors mapped on a resource, the analysis has a high complexity. The iterative P^3 approach computes how much a particular actor contributes to the waiting time of the other actors. This is therefore linear in the number of actors, but needs to be iterated to improve the waiting time estimate. Both techniques compute the expected waiting time when multiple tasks share a processing resource. (The approach can be adapted for other types of resource like communication and memory as well.) These waiting time estimates, together with the execution time are used to estimate the performance of applications. The approach is very fast and can be used both at design-time and run-time owing to its low implementation complexity. Some of the material in this chapter is published earlier in (Kumar et al. 2010).

Following are the key features of the proposed P^3 algorithm.

- *Accurate:* The performance values predicted vary from the measured values by 2% on average and 3% at maximum, as observed.
- *Fast:* The algorithm has the complexity of $O(n)$, where n is the number of actors on each processor.
- *Scalable:* The algorithm is scalable in the number of actors per application, the number of processing nodes, and the number of applications in the system. This implies that when the number of actors or processing nodes are doubled, the execution time for the algorithm is also doubled.

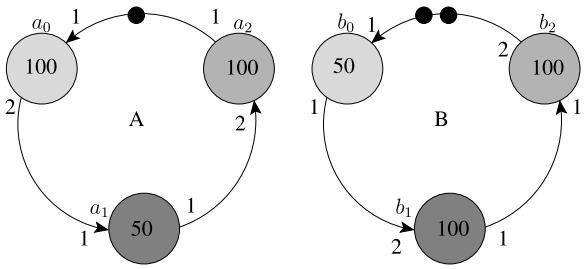


Fig. 3.2 Two application SDFGs *A* and *B*

- *Composable:* The above algorithm uses limited information from the other applications, thereby keeping the entire analysis composable.

The above features make the algorithm very suitable for implementation in embedded multimedia systems.

The remainder of the chapter is organized as follows. Section 1 explains the probabilistic approach that is used to predict performance of multiple applications accurately. Section 2 explains the iterative probabilistic technique that builds upon the probability technique to improve the accuracy of the technique further. Section 3 describes the experimental setup and results obtained. Section 4 gives some pointers to references explaining how performance analysis is done using SDF graphs traditionally – for single and multiple applications. Section 5 presents major conclusions and gives directions for future work.

1 Basic Probabilistic Analysis

When multiple applications execute in parallel, they often cause contention for the shared resources. The probabilistic mechanism predicts this contention. The time spent by an actor in contention is added to its execution time, and the total gives its response time. The equation below puts it more clearly.

$$t_{resp} = t_{exec} + t_{wait} \tag{3.1}$$

The t_{wait} is the time that is spent in contention when waiting for a processor resource to become free. The response time, t_{resp} indicates how long it takes to process an actor after it arrives on a node. When there is no contention, the response time is simply equal to the execution time. Using only the execution time gives us the maximum throughput that can be achieved with the given mapping. At design-time, since the run-time application-mix is not always known, it is not possible to accurately predict the waiting-time, and hence the performance. In this section, we explain how this estimate is obtained using probability. (Some of the definitions used here are explained in Sect. 4 of Chap. 2, and the reader is advised to refer to them, if needed.)

We now refer to SDFGs *A* and *B* in Fig. 3.2. Say a_0 and b_0 are mapped on a processor $Proc_0$. a_0 is active for time $\tau(a_0)$ every $Per(A)$ time units (since its repetition entry is 1). $\tau(a_0) = 100$ time units and $Per(A) = 300$ time units on average.

Assuming the process of executing tasks is stationary and ergodic, the probability of finding $Proc_0$ in use by a_0 at a random moment in time equals $\frac{1}{3}$. We now assume that the arrivals of a_0 and b_0 are stationary and independent; thus the probability of $Proc_0$ being occupied when b_0 arrives is also $\frac{1}{3}$. (We know that in reality these are not independent since there is a dependence on resources. This assumption is made in order to simplify the analysis and keeping it composable. We study the impact of this assumption on the accuracy of the prediction made by this probabilistic model in Sect. 3.) Further, since b_0 can arrive at any arbitrary point during execution of a_0, the time a_0 takes to finish after b_0 arrives on the node, given the fact that a_0 is executing, is uniformly distributed from [0, 100]. Therefore, the expected waiting time is 50 time units and b_0 has to wait for 50 time units on average on a long-run execution whenever it finds $Proc_0$ blocked due to a_0. Since the probability that the resource is occupied is $\frac{1}{3}$, the average time actor b_0 has to wait is given by $\frac{50}{3} \approx 16.7$ time units. The average response time of b_0 will therefore be 66.7 time units.

Generalizing the Analysis

This sub-section generalizes the analysis presented above. As we can see in the above analysis, each actor has two attributes associated with it: (1) the probability that it blocks the resource and (2) the average time it takes before freeing up the resource it is blocking. In view of this, we define the following terms:

Definition 10 (Blocking Probability) Blocking Probability, $P(a)$ is defined as the probability that actor a of application A blocks the resource it is mapped on. $P(a) = \tau(a).q(a)/Per(A)$. $P(a)$ is also represented as P_a interchangeably. $\tau(a)$, $q(a)$ and $Per(A)$ are defined in Chap. 2 as Definitions 5, 7 and 8 respectively.

$P(a_0) = \frac{1}{3}$ in Fig. 3.2.

Definition 11 (Average Blocking Time) Average Blocking Time, $\mu(a)$ is defined as the average time before the resource blocked by actor a is freed given the resource is found to be blocked. $\mu(a)$ is also represented as μ_a interchangeably. $\mu(a) = \tau(a)/2$ for constant execution time, and uniform distribution, i.e. there is no correlation between a and other actors mapped on the same processor.

$\mu(a_0) = 50$ in Fig. 3.2.

Suppose actor b is mapped on processor $Proc_0$, which is also shared by a. If X denotes how long actor b has to wait *when* $Proc_0$ *is being blocked* by actor a, the probability density function, $w(x)$ of X can be defined as follows.

$$w(x) = \begin{cases} 0, & x \leq 0 \\ \frac{1}{\tau(a)}, & 0 < x \leq \tau(a) \\ 0, & x > \tau(a) \end{cases} \tag{3.2}$$

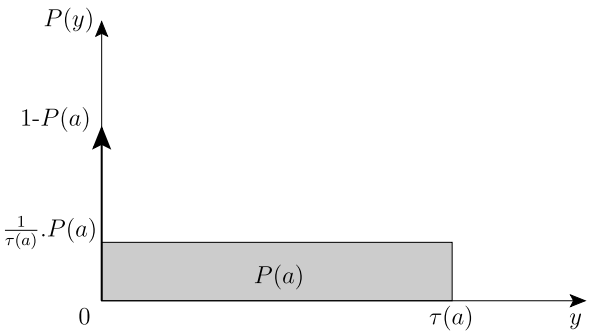


Fig. 3.3 Probability distribution of the time another actor has to wait when actor a is mapped on the resource

The average time b has to wait when the resource is blocked, or μ_a is therefore,

$$\begin{aligned}\mu_a = E(X) &= \int_{-\infty}^{\infty} x\, w(x)\, dx \\ &= \int_{0}^{\tau(a)} x \frac{1}{\tau(a)}\, dx \\ &= \frac{1}{\tau(a)}\left[\frac{x^2}{2}\right]_0^{\tau(a)} \\ &= \frac{\tau(a)}{2} \end{aligned} \tag{3.3}$$

Figure 3.3 shows the overall probability distribution of another actor b waiting for a resource that is shared with a. Here, Y denotes the time actor b has to wait for resource $Proc_0$ after it is ready regardless of whether $Proc_0$ is blocked or not. This includes a δ-function of value $1 - P(a)$ at the origin since that is the probability of $Proc_0$ being available (not being occupied by a) when b wants to execute. Clearly, the total area under the curve is 1, and the expected value of the distribution gives the overall average expected waiting time of b per execution of b and can be computed as

$$t_{wait}(b) = E(Y) = \frac{\tau(a)}{2}.P(a) = \mu_a.P_a \tag{3.4}$$

Let us revisit our example in Fig. 3.2. Let us now assume actors a_i and b_i are mapped on $Proc_i$ for $i = 0, 1, 2$. The blocking probabilities for actors a_i and b_i for $i = 0, 1, 2$ are

$$P(a_i) = \frac{\tau(a_i).q(a_i)}{Per(A)} = \frac{1}{3} \quad \text{for } i = 0, 1, 2$$

$$P(b_i) = \frac{\tau(b_i).q(b_i)}{Per(B)} = \frac{1}{3} \quad \text{for } i = 0, 1, 2$$

The average blocking time of actors (if they are blocked) in Fig. 3.2 is

$$[\mu_{a_0}\ \mu_{a_1}\ \mu_{a_2}] = [50\ 25\ 50] \quad \text{and} \quad [\mu_{b_0}\ \mu_{b_1}\ \mu_{b_2}] = [25\ 50\ 50]$$

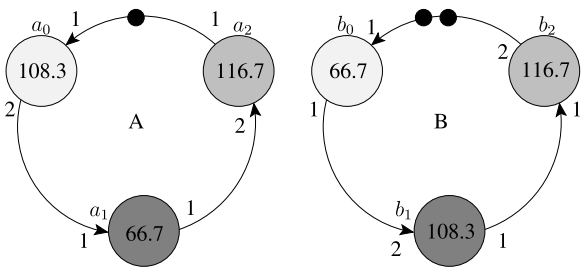


Fig. 3.4 SDFGs *A* and *B* with response times

In this case, since only one other actor is mapped on every node, the waiting time for each actor is easily derived.

$$t_{wait}(b_i) = \mu(a_i).P(a_i) \quad \text{and} \quad t_{wait}(a_i) = \mu(b_i).P(b_i)$$

$$t_{wait}[b_0\ b_1\ b_2] = \left[\frac{50}{3}\ \frac{25}{3}\ \frac{50}{3}\right] \quad \text{and} \quad t_{wait}[a_0\ a_1\ a_2] = \left[\frac{25}{3}\ \frac{50}{3}\ \frac{50}{3}\right]$$

Figure 3.4 shows the response time of all actors taking waiting times into account. The new period of SDFG *A* and *B* is computed as 358.4 time units for both. In practice, the period that these application graphs would achieve is actually 300 time units. However, it must be noted that in our entire analysis we have ignored the intra-graph actor dependency. For example, if the cyclic dependency of SDFG *B* was changed to clockwise, all the values computed above would remain the same while the period of the graphs would change. The period then becomes 400 time units (see Fig. 2.14). The probabilistic estimate we have now obtained in this simple graph is roughly equal to the mean of the periods obtained in either of the cases.

Further, in this analysis we have assumed that arrivals of actors on a node are independent. In practice, this assumption is not always valid. Resource contention will inevitably make the independent actors dependent on each other. Even so, the approach works very well, as we shall see in Sect. 3. A rough sketch of the algorithm used in our approach is outlined in Algorithm 1.

Extending to N Actors

Let us assume actors a, b and c are mapped on the same node, and that we need to compute the average waiting time for c. c may be blocked by either a or b or both. Analyzing the case of c being blocked by both a and b is slightly more complicated. There are two sub-cases – one in which a is being served and b is queued, and another in which b is being served and a is queued. We therefore have four possible cases outlined below, including the waiting time for each case.

Blocking only by a:

$$t_{wait}(c_1) = \mu_a.P_a.(1 - P_b)$$

Algorithm 1: UpdatePeriod: Computing the new period for each application using blocking probabilities

Input: $\tau(a_{ij})$, $q(a_{ij})$, $Per(A_i)$ // Execution time, repetition entry, and original period.
Output: $Per(A_i)$ // Updated Period

```
1: // a_ij is actor j of application A_i
2: for all actors a_ij do
3:    P(a_ij) = BlockingProb(τ(a_ij), q(a_ij), Per(A_i))
4: end for
5: // Now use this to compute waiting time.
6: for all Applications A_i do
7:    for all Actors a_ij of A_i do
8:       t_wait(a_ij) = WaitingTime(τ, P)
9:       τ(a_ij) = τ(a_ij) + t_wait(a_ij)
10:   end for
11:   Per(A_i) = NewPeriod(A_i)
12: end for
```

Blocking only by b:

$$t_{wait}(c_2) = \mu_b.P_b.(1 - P_a)$$

a being served, b queued:
The average time spent by b in each execution of b waiting behind a is given by $\mu_a.P_a$. Therefore, the total probability of b behind a is,

$$P_{wait}(c_3) = \mu_a.P_a.\frac{q[b]}{Per(b)} = P_a.P_b.\frac{\mu_a}{2.\mu_b} \tag{3.5}$$

and the corresponding waiting time is,

$$t_{wait}(c_3) = P_a.P_b.\frac{\mu_a}{2.\mu_b}.(\mu_a + 2\mu_b)$$

b being served, a queued:
This can be derived similar to above as follows:

$$t_{wait}(c_4) = P_b.P_a.\frac{\mu_b}{2.\mu_a}.(\mu_b + 2\mu_a)$$

The time that c needs to wait when two actors are in the queue varies depending on which actor is being served. For example, when a is ahead in the queue, c has to wait for μ_a due to a, since a is being served. However, since the whole actor b remains to be served after a is finished, c needs to wait $2.\mu_b$ for b. One can also observe that the waiting time due to actor a is $\mu_a.P_a$ when it is in front, and $2.\mu_a.P_a$ when behind. Adding the contributions from each of the four cases above to the waiting time, we get

$$t_{wait}(c) = \mu_{ab}.P_{ab} = \frac{1}{2}.P_a.P_b.\left(\frac{\mu_a^2}{\mu_b} + \frac{\mu_b^2}{\mu_a}\right) + \mu_a.P_a + \mu_b.P_b$$

Table 3.1 Probabilities of different queues with a

Queue	Probability	Extra waiting probability
a	$P_a(1-P_b)(1-P_c)$	
ab	$P_a.P_b(1-P_c)/2$	
ba	$P_a.P_b(1-P_c)/2$	$P_aP_b(1-P_c)/2$
ac	$P_a.P_c(1-P_b)/2$	
ca	$P_a.P_c(1-P_b)/2$	$P_aP_c(1-P_b)/2$
abc-acb	$P_a.P_b.P_c/3$	
bca-cba bac-cab	$\frac{2}{3}P_a.P_b.P_c$	$\frac{2}{3}P_a.P_b.P_c$
Total		$\frac{1}{2}P_a(P_b+P_c)-\frac{1}{3}P_aP_b.P_c$

$$= \mu_a.P_a.\left(1+\frac{\mu_a}{2\mu_b}P_b\right)+\mu_b.P_b.\left(1+\frac{\mu_b}{2\mu_a}P_a\right)$$

We observe that the probability terms (that are often < 1) are multiplied. To make the analysis easier, we therefore assume that the probability of a behind b, and b behind a are nearly equal (which becomes even more true when tasks are of equal granularity, since then $\mu_a \approx \mu_b$. This assumption is not needed for the iterative analysis.) Therefore, the above equation can be approximated as,

$$\begin{aligned} t_{wait}(c) &= \frac{1}{2}.P_a.P_b.(\mu_a+\mu_b)+\mu_a.P_a+\mu_b.P_b \\ &= \mu_a.P_a.\left(1+\frac{1}{2}P_b\right)+\mu_b.P_b.\left(1+\frac{1}{2}P_a\right) \end{aligned}$$

The above can be also computed by observing that whenever an actor a is in the queue, the waiting time is simply $\mu_a.P_a$, i.e. the product of the probability of a being in the queue (regardless of other actors) and the waiting time due to it. However, when it is behind some other actor, there is an extra waiting time μ_a, since the whole actor a has to be executed. The probability of a being behind b is $\frac{1}{2}.P_a.P_b$ (from Eq. 3.5) and hence the total waiting time due to a is $\mu_a.P_a.(1+\frac{1}{2}P_b)$. The same follows for the contribution due to b.

For three actors waiting in the queue, it is best explained using a table. Table 3.1 shows all the possible states of the queue with a in it. The first column contains the ordering of actors in the queue, where the leftmost actor is the first one in the queue. All the possibilities are shown together with their probabilities. There are six queues with three actors, and the probability for each of them is approximated as a sixth. For the cases when a is not in front, the waiting time is increased by $\mu_a.P_a$ since the waiting time contributed by a is $2\mu_a$, and therefore, those probability terms are added again. The same can be easily derived for other actors too. We therefore obtain the following equation.

$$\mu_{abc}.P_{abc} = \mu_a.P_a.\left(1+\frac{1}{2}(P_b+P_c)-\frac{1}{3}P_b.P_c\right)$$

$$+ \mu_b.P_b.\left(1 + \frac{1}{2}(P_a + P_c) - \frac{1}{3}P_a.P_c\right)$$
$$+ \mu_c.P_c.\left(1 + \frac{1}{2}(P_a + P_b) - \frac{1}{3}P_a.P_b\right) \tag{3.6}$$

We can also understand intuitively, to some extent, how the terms in the equation derived above contribute to the delay (extra waiting time) in the analysis. The first term of each of the three lines (for instance $\mu_a.P_a$ in first line) denotes the delay due to the respective actors. The terms that follow are the probabilities of the actor being in front of the queue; being there with at least one more actor but behind; and then with at least two more actors and so on and so forth. Since the third probability term (≥ 2 *actors*) is included in the second probability term (≥ 1 *actor*), the last term is subtracted. Similar analysis and reasoning gives us the following equation for waiting time when n actors $a_1, a_2, \ldots a_n$ are mapped on a particular resource

$$\mu_{a_1 \ldots a_n} P_{a_1 \ldots a_n} = \sum_{i=1}^{n} \mu_{a_i} P_{a_i} \left(1 + \sum_{j=1}^{n-1} \frac{(-1)^{j+1}}{j+1} \times \prod_{j} (P_{a_1} \ldots P_{a_{i-1}} P_{a_{i+1}} \ldots P_{a_n})\right) \tag{3.7}$$

where

$$\prod_{j}(x_1, \ldots, x_n) = \sum_{1 \leq k_1 < k_2 \cdots < k_j \leq n} (x_{k_1} x_{k_2} \ldots x_{k_j})$$

$\prod_j (x_1, \ldots, x_n)$ is an elementary symmetric polynomial defined in (Terr and Weisstein 2008). In simple terms, it is the summation of all the products of j unique terms in the set $(x_1, \ldots, x_n)$. The number of terms clearly increases exponentially with increasing n. The total number of terms in Eq. 3.7 in the symmetric polynomial is given by $\binom{n-1}{j}$ i.e. $\frac{(n-1)!}{j!(n-1-j)!}$. As the number of actors mapped on a node increases, the complexity of analysis also becomes high. To be exact, the complexity of the above formula is $O(n^{n+1})$, where n is the number of actors mapped on a node. Since this is done for each actor, the overall complexity becomes $O(n^{n+2})$. In the next sub-section we see how this complexity can be reduced.

Reducing Complexity

The complexity of the analysis plays an important role when putting an idea to practice. In this section we shall look at how the complexity can be reduced. First we shall see how the formula can be rewritten in order to improve the complexity without changing the accuracy. Later we observe that higher order probability products start appearing in the equation as the number of actors mapped on a processor is increased. Thus, we provide two approximations to second and fourth-order respectively. One of the main advantages of the probabilistic approach is the run-time

implementation as is explained in Chap. 4. The *composability-based approach* can make the approach even more usable for run-time when applications enter and leave the system.

The total analysis complexity in Eq. 3.7 is $O(n^{n+2})$. Using some clever techniques for implementation the complexity can be reduced to $O(n^2 + n^n)$ i.e. $O(n^n)$. This can be achieved by modifying the equation such that we first compute $\prod_j(P_{a_1}, P_{a_2} \ldots P_{a_n})$ including P_{a_i}. The extra component is then subtracted from the total for each a_i separately.

However, this is still infeasible and not scalable. An important observation that can be made is that higher order terms start to appear in our analysis. The number of these terms in $\prod_j$ in Eq. 3.7 increases exponentially. Since these terms are products of probabilities (being smaller than one), higher order terms can likely be neglected. To limit the computational complexity, we provide a second order approximation of the formula.

$$\mu_{a_1 \ldots a_n} P_{a_1 \ldots a_n} \approx \sum_{i=1}^{n} \mu_{a_i} P_{a_i} \left(1 + \frac{1}{2} \sum_{j=1,\, j \neq i}^{n} (P_{a_j})\right)$$

The complexity of the above formula is $O(n^3)$, since we have to do it for n actors. For the above equation, we can modify the summation inside the loop such that the complexity is reduced. The new formula is re-written as

$$\mu_{a_1 \ldots a_n} P_{a_1 \ldots a_n} \approx \sum_{i=1}^{n} \mu_{a_i} P_{a_i} \left(1 + \frac{1}{2}(Tot\_Summ - P_{a_i})\right) \tag{3.8}$$

where

$$Tot\_Summ = \sum_{j=1}^{n} P_{a_j}$$

This makes the overall complexity $O(n^2)$. In general, the complexity can be reduced to $O(n^m)$ for $m \geq 2$ by using m-th order approximation. In Sect. 3 we present results of second and fourth order approximations.

Composability-Based Approach

In a system often applications are added at run-time. In the earlier approach if we already have a prediction for a particular combination of applications executing, and another application wants to start, the entire analysis has to be repeated. Here we present an alternative approach, in which when the prediction of a particular combination is already known, the effect of the new application can be simply added to the overall effect of the previous combination. This approach is defined as *composability-based approach*. In this approach the effect of multiple actors together is treated as if it were one actor. If there are two actors, they are *composed* into one actor such that the properties of this new actor can be approximated by the sum of their individual properties. In particular, if we have two actors a and b, we

would like to know their combined blocking probability P_{ab}, and combined waiting time due to them, $\mu_{ab}.P_{ab}$. Thus, when new applications are started, the analysis can be incremental instead of repeating the whole analysis. We further define this composability operation for probability by $\oplus$ and for waiting time by $\otimes$. We therefore get,

$$P_{ab} = P_a \oplus P_b = P_a + P_b - P_a.P_b \tag{3.9}$$

$$\mu_{ab}.P_{ab} = \mu_a.P_a \otimes \mu_b.P_b = \mu_a.P_a.\left(1 + \frac{P_b}{2}\right) + \mu_b.P_b.\left(1 + \frac{P_a}{2}\right) \tag{3.10}$$

(Strictly speaking $\otimes$ operation also requires individual probabilities of the actors as inputs, but this has been omitted in the notation for simplicity.) Associativity of $\oplus$ is easily proven by showing $P_{abc} = P_{ab} \oplus P_c = P_a \oplus P_{bc}$. Operation $\otimes$ is associative only to the second order approximation. This can be proven in a similar way by showing $\mu_{abc}P_{abc} = \mu_{ab}P_{ab} \otimes \mu_c P_c = \mu_a P_a \otimes \mu_{bc}P_{bc}$.

The associative property of these operations reduces the complexity even further. Complexity of Eqs. 3.9 and 3.10 is clearly $O(1)$. If the waiting time of a particular actor is to be computed, all the other actors have to be combined giving a total complexity of $O(n^2)$, which is equivalent to the complexity of the second-order approximation approach. However, in this approach the effect of actors is incrementally added. Therefore, when a new application has to be added to the analysis and new actors are added to the nodes, the complexity of the computation is $O(n)$ as compared to $O(n^2)$ in the case of the second-order approximation, for which the entire analysis has to be repeated.

Computing Inverse of Formulae

The complexity of this *composability-based* approach can be further reduced when we can compute the inverse of the formulae in Eqs. 3.9 and 3.10. When the inverse function is known, all the actors can be *composed* into one actor by deriving their total blocking probability and total average blocking time. To compute the individual waiting time, only the inverse operation with their own parameters has to be performed. The total complexity of this approach is $O(n) + n.O(1) = O(n)$. The inverse is also useful when applications enter and leave the analysis, since only an incremental *add* or *subtract* has to be done to update the waiting time instead of computing all the values. For example, if 10 applications are concurrently executing in the system, and one of them leaves. Normally, we would need to recompute the effect of the remaining 9 applications all over again. However, with the inverse, this can be directly computed. In Eq. 3.9, a then refers to the nine applications and b to the leaving application. Since P_{ab} and P_b i.e. the equivalent probability of all 10 applications and of the leaving application is known, P_a can be computed with the inverse of $\oplus$ operation.

The inverse for both operations is given below. (Note that the inverse formula can only be applied when $P_b \neq 1$.)

$$P_{a_1 \ldots a_n b} = P_{a_1 \ldots a_n} \oplus P_b$$
$$\Rightarrow P_{a_1 \ldots a_n} = P_{a_1 \ldots a_n b} \oplus^{-1} P_b = \frac{P_{a_1 \ldots a_n b} - P_b}{1 - P_b} \quad (P_b \neq 1)$$

$$\mu_{a_1 \ldots a_n b} P_{a_1 \ldots a_n b} = \mu_{a_1 \ldots a_n} P_{a_1 \ldots a_n} \otimes \mu_b P_b$$
$$\Rightarrow \quad \mu_{a_1 \ldots a_n} P_{a_1 \ldots a_n} = \mu_{a_1 \ldots a_n b} P_{a_1 \ldots a_n b} \otimes^{-1} \mu_b P_b$$
$$\Rightarrow \quad \mu_{a_1 \ldots a_n} P_{a_1 \ldots a_n} = \frac{\mu_{a_1 \ldots a_n b} P_{a_1 \ldots a_n b} - \mu_b . P_b \left(1 + \frac{P_{a_1 \ldots a_n}}{2}\right)}{1 + \frac{P_b}{2}}$$

2 Iterative Analysis

So far we have seen the basic analysis. While the basic analysis gives good results (as we see in Sect. 3), we present a technique that can improve them even further. The iterative analysis takes advantage of two facts observed in the previous section.

- An actor contributes to the waiting time of another actor in two ways – while it is being executed, and while it is waiting for the resource to become free.
- The application behaviour itself changes when executing concurrently with other applications. In particular the period of the application changes (increases as compared to the original period) when executing concurrently with interfering applications.

The increase in application period implies that the actors request the resource less frequently than analyzed in the earlier analysis. The application period as defined in Definition 8 is modified due to the difference in actor response times leading to a change in the actor blocking probability. Further, an actor can block another actor in two ways. Therefore, we define two different blocking probabilities.

Definition 12 (Execution Blocking Probability) Execution Blocking Probability, $P_e(a)$, is defined as the probability that actor a of application A blocks the resource it is mapped on, *and is being executed*. $P_e(a) = \tau(a).q(a)/Per_{New}(A)$.

$P_e(a_0) = \frac{100}{358}$ in Fig. 3.4, since $Per_{New}(A) = 358$.

Definition 13 (Waiting Blocking Probability) Waiting Blocking Probability, $P_w(a)$, is defined as the probability that actor a of application A blocks the resource it is mapped on *while waiting for it to become available*. $P_w(a) = t_{wait}(a).q(a)/Per_{New}(A)$.

$P_w(a_0) = \frac{8}{358}$ in Fig. 3.4.

When an actor arrives at a particular processor, it can either find a particular other actor being served, waiting in the queue, or not in the queue at all. If an actor arrives

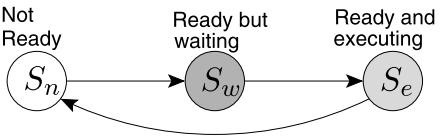


Fig. 3.5 Different states an actor cycles through

when the other actor is waiting, then it has to wait for the entire execution time of that actor (since it is queued at the end). On the other hand when the other actor is being served, the average waiting time due to that actor is half of the total execution time as shown in Eq. 3.3.

There is a fundamental difference with the analysis presented in Sect. 1. In the earlier analysis, an actor had two states – requesting a resource and not requesting a resource. In this analysis, there are three states – waiting in queue on the resource, executing on the resource and not requesting it at all. This explicit state of waiting for the resource, combined with the updated period, represents the blocking effect on another actor more accurately, and also makes understanding the analysis easier.

Figure 3.5 shows that any actor of an SDF graph has to go through three different states. When the actor does not have enough input data or output space i.e. sufficient tokens on all of its incoming edges and available buffer capacity on all of its output edges, it is not ready. This state is denoted by S_n. When the data is available, the actor becomes ready. However, if the required resource is busy then the actor may still have to wait. We denote this state of ready but waiting for the resource to become available as S_w. When the processor or another resource becomes available, the actor starts executing and this state is denoted as S_e.

For an actor whose execution time is constant, the time spent in the executing state S_e does not change, and is simply equal to its execution time $\tau(a)$. The time spent during waiting state S_w depends on the available resources. If there is no other actor mapped on a particular resource, then this time is simply zero. The time spent during not-ready state S_n depends on the graph structure and the period of the graph.

We can define the state of the task (Fig. 3.5) as a stochastic process $S(t)$. We assume that this process is ergodic and stationary. The total probabilities of finding an actor in any of these states is clearly 1. Thus we obtain,

$$P\big(S(t) = S_n\big) + P\big(S(t) = S_w\big) + P\big(S(t) = S_e\big) = 1 \tag{3.11}$$

where $S(t)$ denotes the state at time t. We will see that the steady state probabilities of an actor being in the states described above can be computed by considering the graph structure, the actor execution time and some properties of other actors mapped on the sharing resource. The probability of finding an actor a in executing state S_e can be computed by considering how often it executes i.e. its repetition vector entry $q(a)$, and its execution time $\tau(a)$. To put it precisely, the actor a executes $q(a)$ times every period $Per(A)$ of the application A to which a belongs, and each time it spends $\tau(a)$ cycles in the state S_e. Thus, the total time spent is $q(a).\tau(a)$ every $Per(A)$. Thus, because of the stationarity of the process, the steady state probability of finding actor a in the executing state is given by the following equation.

$$P\big(S(t) = S_e\big) = \frac{q(a).\tau(a)}{Per(A)} \tag{3.12}$$

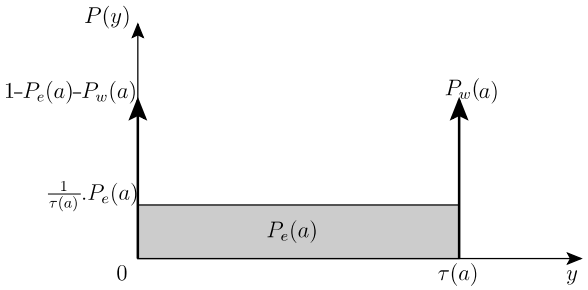


Fig. 3.6 Probability distribution of the waiting time added by actor a to other actor when actor a is mapped on the resource with explicit waiting time probability

When the actor is sharing resources with other actors it may also have to wait for the resource to become available. If the average waiting time is denoted by $t_{wait}(a)$, then the total time spent in the waiting state, on average, is given by $q(a).t_{wait}(a)$ every $Per(A)$. Thus, the steady state probability of finding actor a in the waiting state is given by the following equation.

$$P\big(S(t) = S_w\big) = \frac{q(a).t_{wait}(a)}{Per(A)} \tag{3.13}$$

Since the total probability for all the states should be 1, the probability of actor a being in the non-ready state can be computed as follows:

$$P\big(S(t) = S_n\big) = 1 - \frac{q(a).t_{wait}(a)}{Per(A)} - \frac{q(a).\tau(a)}{Per(A)} \tag{3.14}$$

Assuming that the arrival time of b is completely independent of the different states of a, the probability of b finding a in a particular state is simply the stationary probability of a being in that state. Further, our assumption also implies that when b arrives and finds a in a particular state, a may be anywhere, with uniform distribution, in that state. Thus, if b finds a in the S_e state, then the remaining execution time is uniformly distributed. Since the probability of finding a in a particular state is directly related to the waiting time of b, we obtain the probability distribution for waiting time of b as shown in Fig. 3.6.

As shown in Fig. 3.6 the time actor b has to wait depends on the state of actor a when b arrives. When b arrives in the S_w state of a, it has to always wait for $\tau(a)$. This gives the δ-function of $P_w(a)$ at $\tau(a)$. On the other extreme we have the δ-function at origin due to b arriving in the S_n state of a. The probability of this is simply equal to the probability of a being in this state, as mentioned earlier. In the middle we have a uniform distribution with the total probability of $P_e(a)$, i.e. a being in S_e state.

If Y denotes how long actor b has to wait for the resource it shares with actor a, the probability density function, $P(y)$ of Y can be defined as follows:

$$P(y) = \begin{cases} 0, & y < 0 \\ \delta(y).(1 - P_e(a) - P_w(a)), & y = 0 \\ \frac{1}{\tau(a)}.P_e(a), & 0 < y < \tau(a) \\ \delta(y - \tau(a)).P_w(a), & y = \tau(a) \\ 0, & y > \tau(a) \end{cases} \tag{3.15}$$

The average waiting time due to actor a for b, $E(Y)$ can now be computed as follows:

$$\begin{aligned} E(Y) &= \int_{-\infty}^{\infty} y P(y)\, dy \\ &= \int_{0}^{\tau(a)} y \frac{1}{\tau(a)}.P_e(a)\, dy + \tau(a).P_w(a) \\ &= \frac{1}{\tau(a)} P_e(a) \left[\frac{y^2}{2}\right]_0^{\tau(a)} + \tau(a).P_w(a) \\ &= \frac{\tau(a)}{2} P_e(a) + \tau(a).P_w(a) \\ &= \tau(a)\left(\frac{P_e(a)}{2} + P_w(a)\right) \end{aligned} \tag{3.16}$$

If $\tau(a)$ is not constant but varying, $E(Y)$ also varies with $\tau(a)$. In such cases, $E(Y)$ can be computed as follows:

$$\begin{aligned} E(Y) &= E\left(\tau(a)\left(\frac{P_e(a)}{2} + P_w(a)\right)\right) \\ &= E\big(\tau(a)\big)\left(\frac{P_e(a)}{2} + P_w(a)\right) \end{aligned} \tag{3.17}$$

Thus, an actor with variable execution time within a uniform distribution is equivalent to an actor with a constant execution time, equal to the mean execution time. (It is equivalent only in terms of its expected value, not of its distribution.) If $\tau(a)$ is uniformly distributed between $\tau_{min}(a)$ and $\tau_{max}(a)$, the overall average waiting time is as given below:

$$E(Y) = \left(\frac{\tau_{min}(a) + \tau_{max}(a)}{2}\right)\left(\frac{P_e(a)}{2} + P_w(a)\right) \tag{3.18}$$

Since Eq. 3.16 represents the waiting time of one actor due to another actor, when there are more actors mapped on a resource each of the mapped actors causes a given actor to wait. If we have 3 actors – a, b, and c, mapped to the same resource, and the waiting time of actor c has to be computed, the following formula can be derived which includes the waiting time due to both a and b.

$$t_{wait}(c) = \frac{\tau_a}{2}.P_e(a) + \tau_a.P_w(a) + \frac{\tau_b}{2}.P_e(b) + \tau_b.P_w(b)$$

Taking the example above as shown in Fig. 3.4, the new periods as computed from the probabilistic analysis in earlier section are 358.4 time units for both A and B. Thus, the new blocking probabilities are obtained as follows:

$$P_e[a_0\, a_1\, a_2] = \left[\frac{100}{358}\ \frac{100}{358}\ \frac{100}{358}\right], \qquad P_e[b_0\, b_1\, b_2] = \left[\frac{100}{358}\ \frac{100}{358}\ \frac{100}{358}\right]$$

$$P_w[a_0\, a_1\, a_2] = \left[\frac{8}{358}\ \frac{34}{358}\ \frac{17}{358}\right], \qquad P_w[b_0\, b_1\, b_2] = \left[\frac{34}{358}\ \frac{8}{358}\ \frac{17}{358}\right]$$

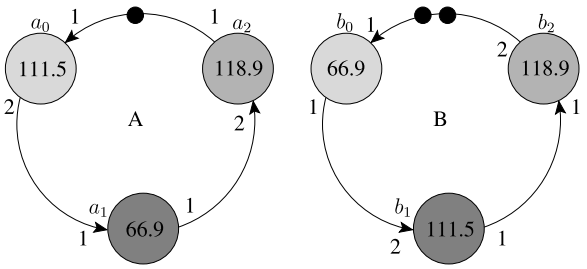


Fig. 3.7 SDF application graphs *A* and *B* updated after applying iterative analysis technique

This gives the following waiting time estimates.

$$t_{wait}[a_0\ a_1\ a_2] = [11.7\ 16.2\ 18.6] \quad \text{and}$$
$$t_{wait}[b_0\ b_1\ b_2] = [16.2\ 11.7\ 18.6]$$

The period for both *A* and *B* evaluates to 362.7 time units. Repeating this analysis for another iteration gives the period as 364.3 time units. Repeating the analysis iteratively gives 364.14, 364.21, 364.19, 364.20, and 364.20 thereby converging at 364.20. Figure 3.7 shows the updated application graphs after the iterative technique is applied.

For a system in which 3 actors – a, b, and c, are mapped to the same node, when waiting time of an actor c has to be computed like above, the following formula can be derived from Fig. 3.6. (Note that $\tau(a) = 2.\mu_a$.)

$$t_{wait}(c) = \mu_a.P_e(a) + 2.\mu_a.P_w(a) + \mu_b.P_e(b) + 2.\mu_b.P_w(b)$$

The above equation shows the beauty of this approach. Unlike the basic approach where the equation becomes complicated with increasing number of actors, this remains a simple addition regardless of how many actors are mapped. For the total waiting time due to n actors, we get the following equation.

$$t_{wait} = \sum_{i=1}^{n} \left(\mu_{a_i} P_e(a_i) + 2\mu_{a_i} P_w(a_i)\right) \tag{3.19}$$

The change in period as mentioned earlier leads to a change in the execution and waiting probabilities of actors. This in turn, changes the response times of actors, which in turn may change the period. This very nature of this technique defines its name *iterative probability*. The cycle is therefore repeated until the periods of all applications stabilise. Figure 3.8 shows the flow for the iterative probability approach. The inputs to this flow are the application throughput expressions, and the execution time and mapping of each actor in all the applications. These, like in the approach mentioned earlier, are first used to compute the base period (i.e. the minimum period without any contention) and the blocking probability of the actor. Using the mapping information, a list of actors is compiled from all the applications and grouped according to their resource mapping. For each processor, the probability analysis is done according to Eq. 3.19. The waiting times thus computed are used again to compute the throughput of the application and the blocking probabilities. The analysis can be run for a fixed number of iterations or terminate using some heuristic as explained below.

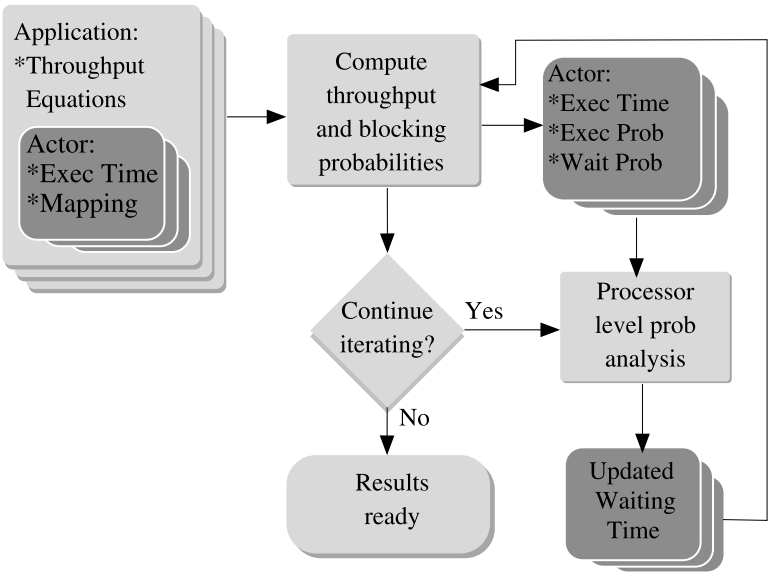


Fig. 3.8 Iterative probability method. Waiting times and throughput are updated until needed

Terminating Condition

While the analysis can be repeated for a fixed number of iterations, it can also be based on the convergence of some parameters. Some candidates for testing convergence are provided below.

- *Application Period:* When the application period for all the applications does not change more than a pre-defined percentage, the analysis can said to have been converged. In our experiments we observed, that just after 6 iterations all applications had a change of less than 1% even when starting from original period. With increasing number of iterations, the variation only became lower. This is the easiest measure since the application period is computed each iteration. The only addition is storing the result of previous iteration, and computing the change.
- *Processor Utilization:* The analysis termination can also be based on the change in processor utilization. The utilization of processors varies with the load predicted by the algorithm. The load on a processor is defined as the sum of the probabilities of execution, $P_e(a)$, of all actors mapped on it. When the algorithm has converged, the load on the processor does not change. Further, the load on the processor determines the waiting time significantly. When the total load on a processor is more than 1, clearly the actors mapped on the processor will have to wait longer. To allow faster convergence, in fact, we scale the waiting time predicted for a particular actor by the total load on the processor it is mapped on.

Fig. 3.9 Probability distribution of waiting time another actor has to wait when actor a is mapped on the resource with explicit waiting time probability for the conservative iterative analysis

Conservative Iterative Analysis

For some applications, the user might be interested in having a more conservative bound on the period i.e. it is better to have a less accurate pessimistic estimate than an accurate optimistic estimate; a much better quality than predicted is more acceptable as compared to even a little worse quality than predicted. In such cases, we provide here a conservative analysis using our iterative technique.

In earlier analysis, when an actor b arrives at a particular resource and finds it occupied by say actor a, we assume that a can be anywhere in the middle of its execution, and therefore, b has to wait on average half of execution time of a. In the conservative approach, we assume that b has to always wait for full execution of a. In the probability distribution as presented in Fig. 3.6, the rectangular uniform distribution of $P_e(a)$ is replaced by another delta function at $\tau(a)$ of value $P_e(a)$. This is shown in Fig. 3.9. The waiting time equation is therefore updated to the following.

$$t_{wait} = \sum_{i=1}^{n} 2\mu_{a_i} \left(P_e(a_i) + P_w(a_i) \right) \tag{3.20}$$

Applying this analysis to the example in Fig. 3.2 starting from the original graph, we obtain the period as 416.7, 408, 410.3, 409.7 and 409.8. Starting from probabilistic analysis values it also stabilises at 409.8 in 5 iterations. Note that in our example, the actual period will be 300 in the best case and 400 in the worst case. The conservative iterative analysis correctly finds the bound of about 410, which is only 2.5% more than the actual worst case. If we apply real worst-case analysis in this approach, then we get a period of 600 time units, which is 50% over-estimated.

This analysis can be either applied from the original period directly, or only after the basic iterative analysis is already converged and terminated. The latter has the benefit of using a realistic period, instead of a conservative period. Since a conservative period is generally higher than the corresponding realistic period, the execution and waiting probability is correspondingly lower when using the conservative period. Thus, using a realistic period with a conservative analysis for the last iteration gives the most conservative results. In experiments below, we present results of both approaches.

Parametric Throughput Analysis

Throughput computation of an SDF graph is generally very time consuming as explained in Chap. 2. Lately, techniques have been presented in (Ghamarian et al. 2006) that can compute throughput of many multimedia applications within milliseconds. However, those results have been taken on a high-end computer while assuming fixed actor execution times. Therefore, throughput computation of an SDF graph is generally done off-line or at design-time for a particular graph. However, if the execution time of an actor changes, the entire analysis has to be repeated. Recently, a technique has been proposed to derive throughput equations for a range of execution times (defined as *parameters*) at design-time and these equations can be easily evaluated at run-time to compute the critical cycle, and hence the period (Ghamarian et al. 2008). This technique greatly enhances the usability of the iterative analysis. With this the iterative analysis can be applied at both design-time and run-time.

For example, for application A shown in Fig. 3.2, there is only one critical cycle. If the execution times of all actors of A are variable, the following parametric equation is obtained (assuming auto-concurrency of 1):

$$Per(A) = \tau(a_0) + 2 \times \tau(a_1) + \tau(a_2) \tag{3.21}$$

Thus, whenever the period of application A is needed, the above equation can be computed with the updated response times of actors a_0, a_1 and a_2. While in this case there is only one equation for application A, in general the number of equations depends on the graph structure and the range of execution times. When there are multiple equations, all of them need to be evaluated to find the limiting period. This technique makes the iterative analysis suitable for run-time implementation.

Intra-task Dependencies

There are two ways of handling the situation when more than one actor of the same application are mapped on the same resource, depending on how it is handled in the real system. One way is to serialize (or order) executions of all actors of a given application. This implies computing a static-order for actors of a given application such that maximum throughput is guaranteed. This can be done using SDF^3 tool (Stuijk et al. 2006a). Once the static-order is computed, the partial order of actors mapped on the same resource can be extracted. The arbiter has to ensure that at any one point in time the actors of an application are executed in this pre-computed order. This ensures that actors of the same application are not queued at the same time. Thus, there is no waiting time added from these actors. For example, in Fig. 3.2 if actors a_0 and a_2 are mapped on the same processor, the static schedule for that processor will be $(a_0 a_2)^*$. A static order adds an extra dependency on actors a_0 and a_2, ensuring that they cannot be ready at the same time, and hence cannot cause

contention for the actors mapped on the same processor. Equation 3.19 for an actor of application A can then be updated for this case as follows:

$$t_{wait} = \sum_{i=1,\, a_i \notin A}^{n} \left(\frac{\tau_{a_i}}{2} P_e(a_i) + \tau_{a_i} P_w(a_i) \right) \tag{3.22}$$

The above approach however, requires extra support from the arbiter. The easiest approach from the arbiter perspective is to treat all the actors mapped on the resource identically and let the actors of the same application also compete with each other for resources. The latter is evaluated in Sect. 3.

Handling Other Arbiters

The above analysis has been presented for first-come-first-serve (FCFS) arbitration. For static-order schedulers like round-robin or another arbitrary order derived from SDF^3 (SDF3 2009), the schedule can be directly modeled in the graph itself. Other dynamic-order schedulers like a priority-based scheduler can be easily modeled in the probability approach. One key difference in a priority-based scheduler as compared to FCFS is that in FCFS once the actor arrives, it has to always wait for actors ahead of it in the queue. In a priority-based system, if it is preemptive, a higher priority actor can immediately preempt a lower priority actor, and if it is non-preemptive, it has only to wait for lower priority actors if they are executing. Let us define the priority of an actor a by $Pr(a)$, such that a higher value of $Pr(a)$ implies a higher priority. Equation 3.19 that is presented for FCFS, can be rewritten as Eq. 3.23. It shows the waiting time for an actor a when sharing a resource with actors a_1 to a_n. Note that the waiting time contributed by the arrival of actor a during the queuing phase of an actor with a priority lower than that of a, is not added in the equation. Similarly, Eq. 3.24 shows the adapted version of Eq. 3.20 for handling priority-based schedulers. It can be seen that these equations are a generalized form of earlier versions, since in FCFS the priorities of all actors are equal, i.e. $Pr(a) = Pr(a_i)\ \forall i = 1, 2, \ldots n$. It should be further noted, that since the priorities are only considered for local analysis on a processor (or any resource), different processors (or resources) can have different arbiters.

$$t_{wait} = \sum_{i=1}^{n} \left(\frac{\tau_{a_i}}{2} P_e(a_i) \right) + \sum_{i=1,\, Pr(a_i) \geq Pr(a)}^{n} \left(\tau_{a_i} P_w(a_i) \right) \tag{3.23}$$

$$t_{wait} = \sum_{i=1}^{n} \left(\tau_{a_i} P_e(a_i) \right) + \sum_{i=1,\, Pr(a_i) \geq Pr(a)}^{n} \left(\tau_{a_i} P_w(a_i) \right) \tag{3.24}$$

3 Experiments

In this section, we describe our experimental setup and some results obtained for the basic probability, explained in Sect. 1. The iterative technique as explained in

Sect. 2 improves upon this. First, we only show results of the basic probability analysis since iterative analysis results are very close to the measured results. Superimposing iterative analysis results on the same scale makes the graph difficult to understand. In the basic analysis results, the graph is scaled to the original period, while in the iterative analysis it is scaled to the measured period. The results of the software implementation of the probability approaches on an embedded processor, Microblaze are also provided.

Setup

In this section we present the results of above analysis obtained as compared to simulation results for a number of use-cases. For this purpose, ten random SDFGs were generated with eight to ten actors each using the SDF^3 tool (Stuijk et al. 2006a), mimicking DSP and multimedia applications. Each graph is a strongly connected component i.e. every actor in the graph can be reached from every actor. The execution time and the rates of actors were also set randomly. The SDF^3 tool was also used to analytically compute the periods of the graphs. Using these ten SDFGs, over a thousand use-cases (2^{10}) were generated. Simulations were performed using POOSL (Theelen et al. 2007) to measure the actual performance for each use-case. Two different probabilistic approaches were used – the second order and the fourth order approximations of Eq. 3.7. Results of worst-case-response-time analysis (Hoes 2004) for non-preemptive systems are also presented for comparison. The worst-case estimate indicates the maximum time an actor may have to wait in a non-preemptive system with the first-come-first-serve mechanism. This estimate is computing using Eq. 2.1.

The simulation of all possible use-cases, each for 500,000 cycles took a total of 23 hours on a Pentium 4 3.4 GHz with 3 GB of RAM. In reality simulation is often done for a lot more cycles. In contrast, analysis for all the approaches was completed in about 10 minutes only.

Results and Discussion – Basic Analysis

Figure 3.10 shows a comparison between periods computed analytically using different approaches as described in this chapter (without the iterative analysis), and the simulation result. The use-case for this figure is the one in which all applications are executing concurrently. This is the case with maximum contention. The period shown in the figure is normalized to the original period of each application that is achieved in isolation. The worst case period observed during simulation is also shown.

A number of observations can be made from this figure. We see that the period is much higher when multiple applications are run. For application C, the period is

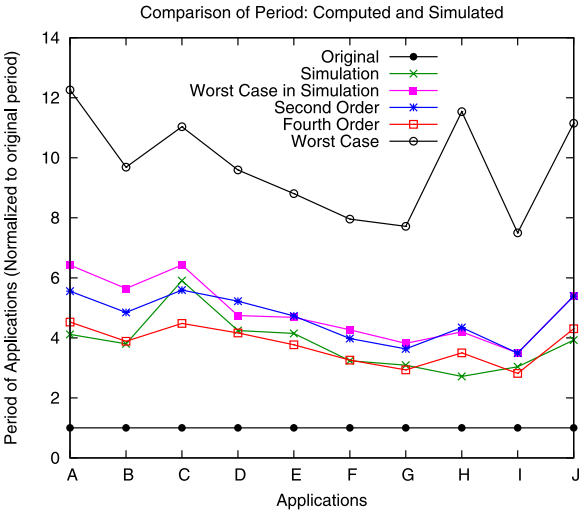


Fig. 3.10 Comparison of periods computed using different analysis techniques as compared to the simulation result (all 10 applications running concurrently). All periods are normalized to the original period

six times the original period, while for application H, it is only three-fold (simulation results). This difference comes from different graph structures and repetition vector entries of actors in different graphs. The analytical estimates computed using different approaches are also shown in the same graph. The estimates using the worst-case-response-time (Bekooij et al. 2005) are much higher than those achieved in practice and therefore, overly pessimistic. The estimates of the two probabilistic approaches are very close to the observed performance.

We further notice that the second order estimate is always more conservative than the fourth order estimate, which is expected, since it overestimates the resource contention. The fourth order estimates of probability are the closest to the simulation results except in applications C and H.

Figure 3.11 shows the variation in period that is estimated and observed as the number of applications executing concurrently in the system increases. The metric displayed in the figure is the mean of absolute differences between estimated and observed periods. This inaccuracy is defined as $\frac{1}{m}\sum_{i=1}^{m}|t_{pred}(a_i) - t_{meas}(a_i)|$, where m is the total number of actors in the system. When there is only one application active in the system, the inaccuracy is zero for all the approaches, since there is no contention. As the number of applications increases, the worst-case-response-time estimate deviates a lot from the simulation result. This indicates why this approach is not scalable with the number of applications in the system. For the other three approaches, we observe that the variation is usually within 20% of the simulation result. We also notice that the second order estimate is very close to the composability-based approach – both of which are more conservative than the fourth-order approximation. The maximum deviation in the fourth order approximation is about 14% as compared to about 160% in the worst-case approach – a ten-fold improvement.

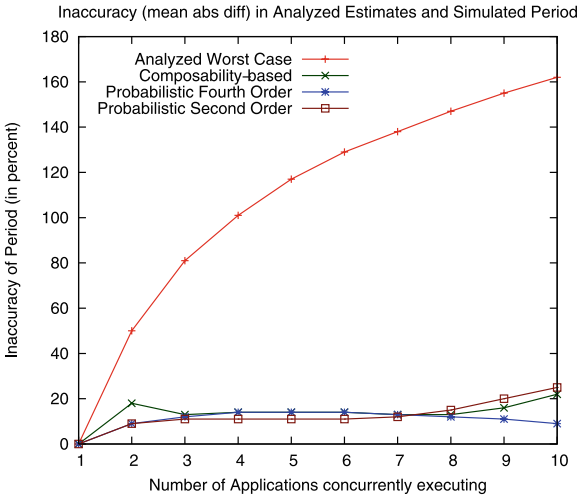


Fig. 3.11 Inaccuracy in application periods obtained through simulation and different analysis techniques

Results and Discussion – Iterative Analysis

Validating the Probabilistic Distribution

In order to check the accuracy of the probabilistic distribution of waiting times presented in Fig. 3.6, we measured exactly when actors arrive when sharing a processor (or another resource) with another actor. For every execution of an actor a, three events are recorded in the processor log file – queuing time (t_q), execution start-time (t_s), and execution end-time (t_e). When other actors arrive between t_q and t_s, they have to wait for the entire execution of a. When they arrive between t_s and t_e, the waiting time depends on where a is in its execution. When the actors arrive between t_e and the next t_q, a does not have any effect on their waiting time. This was measured and summarized for the entire simulation for all the actors. Here we present results of two actors – one randomly chosen from a processor with high utilization, and another with low utilization. This is done in order to check if the model still holds as the utilization of the processor approaches 1. Figure 3.12 shows the distribution of this waiting time for actor $a2$ of application F mapped on processor 2. Processor 2 has a high utilization of almost 1. The distribution is obtained from about three thousand arrivals. This actor takes 35 time units to execute. The distribution of actor arrival times assumed in the model is also shown in the same figure for comparison. A couple of observations can be made from this figure. The distribution between 0 and 35 is more or less uniform. The mean of this uniform distribution observed in the experiment is a bit less than the model. The arrival times of other actors when $a2$ is not in the queue are somewhat higher than that assumed in the model, and the arrivals in the queuing time of $a2$ are rather accurate. If we look at the total waiting time contributed by $a2$, the prediction using the assumed arrival model is 13.92, whereas the measured mean delay contributed by $a2$ is 12.13 – about 15% lower. The conservative assumption predicts the waiting time to be 17.94 due to $a2$.

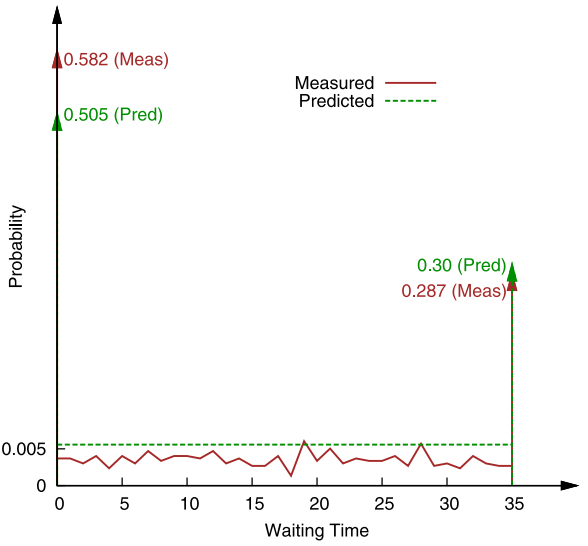


Fig. 3.12 Probability distribution of the time other actors have to wait for actor $a2$ of application F. $a2$ is mapped on processor 2 with a utilization of 0.988. The overall waiting time measured is 12.13, while the predicted time is 13.92. The conservative prediction for the same case is 17.94

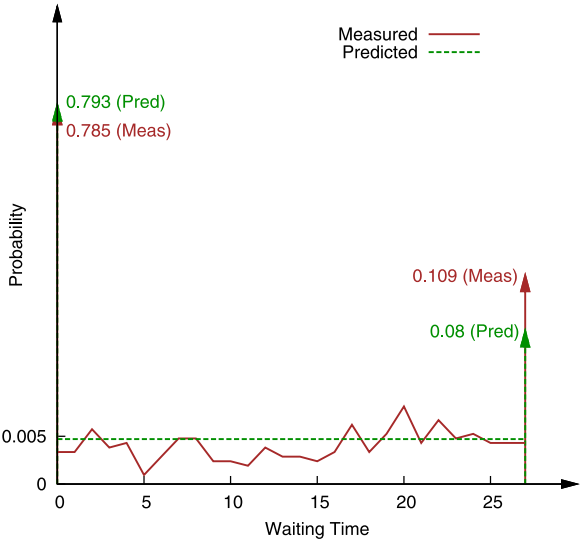


Fig. 3.13 Probability distribution of the time other actors have to wait for actor $a5$ of application G. $a5$ is mapped on processor 5 with a utilization of 0.672. The overall waiting time measured is 4.49, while the predicted time is 3.88. The conservative prediction for the same case is 6.84

Figure 3.13 shows the similar distribution for actor $a5$ of application G mapped on processor 5. This processor has comparatively low utilization of 0.672. Similar to the earlier graph, the probability distribution of the waiting time is uniform between 0 and 27, the execution time of this actor. In this case, while the delta function at 0 is almost equal to the assumed model, the delta function at 27 is almost 35% higher than what we assume in the model. This is quite contrary to the expectation since this processor has lower utilization. This probably happens because of the inter-dependencies that are created by resource arbitration between concurrently

Table 3.2 Comparison of the time actors actually spend in different stages assumed in the model vs the time predicted

Actor	No Request Time		Executing		Queuing	
	Pred	Meas	Pred	Meas	Pred	Meas
F, $a2$	0.505	0.556	0.195	0.189	0.300	0.255
G, $a5$	0.793	0.770	0.127	0.127	0.080	0.102

executing multiple applications. The overall waiting time assumed in the analysis is 3.88, while we measured 4.49 – 15% higher this time. However, the conservative estimate of 6.84 is still higher than 4.49.

Thus, we see that our assumption of the probability distribution in Fig. 3.6 consisting of two delta functions and a uniform distribution in the middle. This is the contribution of one actor to the other actors. The total waiting time of an actor is the combined effect of all the other actors mapped on a resource. We see the variation of the predicted waiting time with the measured waiting time in the following sub-section.

It should be mentioned that above figures show the arrival of the other actors when a particular actor is queued. The distribution of the actor itself in the three stages – queuing, executing, and not requesting at all – is not captured. This behaviour is captured by observing the entire simulation time and the results are summarized in Table 3.2. Looking at this table, it is easy to explain the small discrepancies in the probabilistic distribution in Fig. 3.12 and Fig. 3.13. Actor $a2$ of application F does not request the resource for a longer time than predicted – 0.556 instead of 0.505. Therefore, there are more arrivals than predicted in the period that it is not requesting the resource at all. The same goes for actor $a5$ of application G. The proportion of time $a5$ spends in the *no-request phase* of 0.770 is a little lower than the prediction of 0.793 and the proportion of arrivals follows. Another observation we can make is that the number of arrivals in the queuing phase is somewhat higher than the actual time spent by the actor in it – for $a2$, 0.287 instead of 0.255 and for $a5$, 0.109 instead of 0.102 (from Fig. 3.12 and Fig. 3.13 respectively). This behaviour is explained when we consider the execution behaviour of the actors. On a processor with high utilization (processor 2), as soon as an actor finishes execution, it is often immediately queued again since there are generally sufficient input tokens available for it. Thus, whenever the actor $a2$ starts executing (by virtue of the processor becoming idle), the actor that just finished executing is queued immediately after. This leads to more arrivals that have to wait for the entire execution of $a2$, i.e. 35, and the effect is an increased queuing probability in the queuing phase. On the processor with lower utilization (processor 5), the effect is somewhat reduced.

So far we have seen the contribution of waiting times from individual actors. The whole idea of this model is to compute the total waiting time for a particular actor correctly. Let us now look at the waiting time predictions of the actors as compared to their measured waiting time. Figure 3.14 shows the total waiting time for actors of different applications mapped on Processor 2. The results of the basic

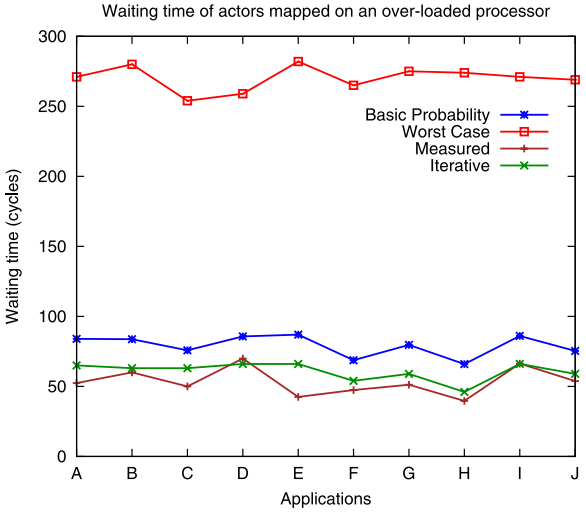


Fig. 3.14 Waiting time of actors of different applications mapped on Processor 2. The utilization of this processor 0.988

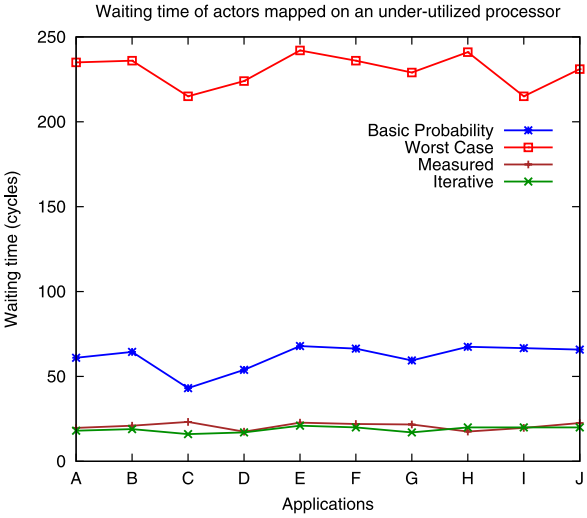


Fig. 3.15 Waiting time of actors of different applications mapped on Processor 5. The utilization of this processor 0.672

probability analysis (fourth order), iterative analysis, and the measured waiting time are presented. The worst case waiting time computed using Eq. 2.1 is shown for comparison. As can be seen, the worst case waiting time computed is much more than that measured on the processor for all applications. The basic analysis brings this waiting time much closer to the measured value, and iterative analysis makes it almost equal to the measured value. This processor has a utilization close to 1. We see that for application E, the iterative analysis predicts a value that is about 40% more than the measured value. However, this is only for one actor of the application (the actor mapped on Processor 2). When the throughput of the whole application is computed, we observe that it is almost equal to the measured value. Figure 3.15

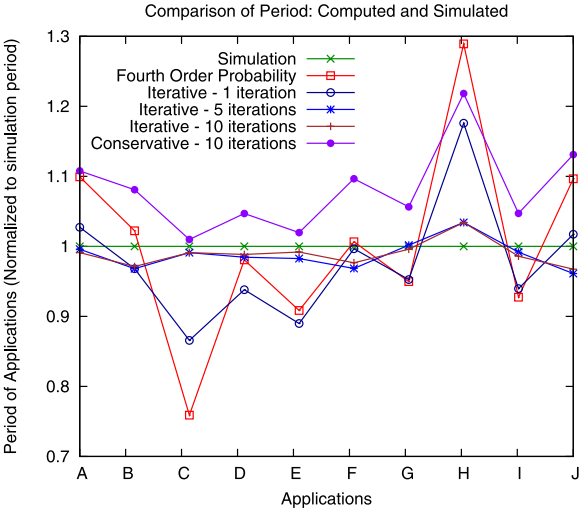


Fig. 3.16 Comparison of periods computed using iterative analysis techniques as compared to simulation results (all 10 applications running concurrently)

shows a similar graph for processor 5. This processor has a lower utilization of 0.672, and the waiting times are also much lower than that guaranteed by the worst case estimate. Also in this case, the basic probability approach gives a slightly higher estimate of waiting time than that measured. Similar trends are observed for other processors as well.

Application Throughput

Figure 3.16 shows the strength of the iterative analysis. The results are now shown with respect to the results achieved in simulation as opposed to the original period. The fourth-order probability results are also shown on the same graph to put things in perspective since that is the closest to the simulation result. As can be seen, while the maximum deviation in fourth-order is about 30%, the average error is very low. The results of applying iterative analysis starting from fourth order, after 1, 5 and 10 iterations are also shown. The estimates get closer to the actual performance after every iteration. After 5 iterations, the maximum error that can be seen in application H is about 3%, and the average error is to the tune of 2%.

Results of the conservative version of the iterative technique are also shown on the same graph. These are obtained after ten iterations of the conservative technique. The estimate provided by this technique is always above the simulation result. On average, in this figure the conservative approach over-estimates the period by about 8% – a small price to pay when compared to the worst-case bound that is 162% over-estimated.

Figure 3.17 shows the results of iterative analysis with an increasing number of iterations for application *A*. Five different techniques are compared with the simulation result – the iterative technique starting from the original graph, the second order

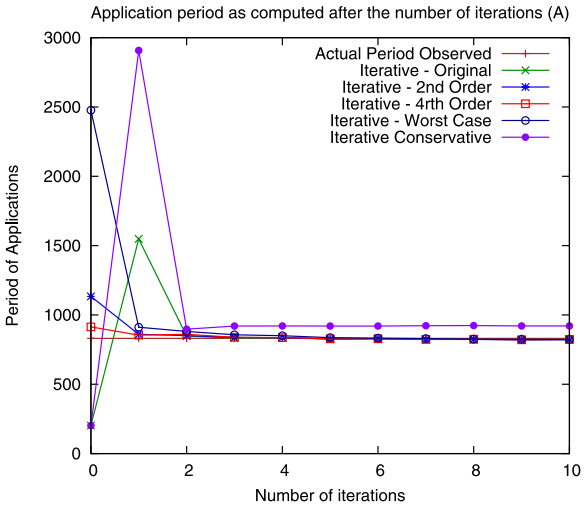


Fig. 3.17 Change in period computed using iterative analysis with increase in the number of iterations for application *A*

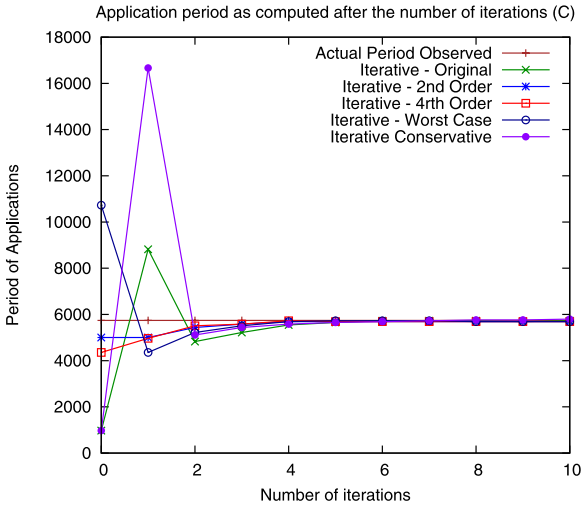


Fig. 3.18 Change in period computed using iterative analysis with increase in the number of iterations for application *C*

probabilistic estimate, the fourth order probabilistic estimate, and the worst case initial estimate, including the conservative analysis of the iterative technique starting from the original graph. While most of the curves converge almost exactly on the simulation result, the conservative estimate converges on a value slightly higher, as expected. A similar graph is shown for another application *C* in Fig. 3.18. In this application it takes somewhat longer before the estimate converges.

A couple of observations can be made from this graph. First, the iterative analysis approach is converging. Regardless of how far and at which side the initial estimate of the application behaviour is, it converges within a few iterations close to the actual value. Second, the final value estimate is independent of the starting estimate.

Table 3.3 Measured inaccuracy for period in % as compared with simulation results for iterative analysis. Both the average and maximum are shown

Iterations	2nd Order	4th Order	Worst Case	Original	Conservative
0	22.3/44.5	9.9/28.9	72.6/83.1	163/325	72.6/83.1
1	6.2/19	6.7/17.6	88.4/144	12.6/36	252/352
2	3.7/13.3	3.5/11.9	6.3/17.6	6.7/23.2	7.9/23.2
3	3/7.7	2.9/6.2	4.5/11.9	4.3/13.3	8.8/24.7
4	2.2/6.2	**2**/4.8	2.5/7.7	3.1/9.1	8.4/23.2
5	2.2/4.8	1.9/3.9	2.2/4.8	2.5/6.2	8.3/23.2
6	**1.7**/3.6	1.6/3.6	**1.7**/3.4	**2**/4.8	8.1/21.8
7	1.8/4	1.9/4	1.8/3.4	1.7/3.9	8/21.8
8	1.7/3.6	1.7/3.6	1.7/3.4	1.8/3.6	8/21.8
9	1.8/3.4	1.9/3.4	1.7/3.6	1.7/3.4	8/21.8
10	1.6/3.3	1.7/3.4	1.3/3.1	1.9/3.4	8.1/21.8
20	1.7/3	1.4/2.9	1.4/2.9	1.5/3	8.1/21.8
30	1.4/3	1.6/3	1.6/3	1.4/3	8.1/21.8

The graph shows that the iterative technique can be applied from any initial estimate (even the original graph directly) and still achieve accurate results. This is a very important observation since this implies that if we have constraints on program memory, we can manage with only the iterative analysis technique. If there is no such constraint, one can always start with the fourth-order estimate in order to get faster convergence. This is probably only suitable for cases when applications have a large number of throughput equations, and when throughput computation takes more cycles than fourth order estimate.

The error in the iterative analysis (defined as mean absolute difference) is presented in Table 3.3. Both the average and the maximum error are shown. Different starting points for the iterative analysis are taken. A couple of observations can be made from the table. Regardless of the starting estimate, the iterative analysis always converges. In general, as the number of iterations increases, the error decreases. As can be seen, the fourth order initial estimate converges the fastest among all approaches. If we define 2% error margin as acceptable, we find that the fourth order estimate requires only 4 iterations to converge while others require 6 iterations. However, obtaining the estimate of the fourth-order analysis is computationally intensive. Using the worst-case or the original period itself as the starting point for the iterative analysis saves the initial computation time, but takes a little longer to converge. When the conservative approach is applied after the base iterative analysis, the average variation is 10% and the maximum error is 16%. Another observation we can make is that in general, there is not much change after 5 iterations. Thus, 5 iterations present a good compromise between the accuracy and the execution time.

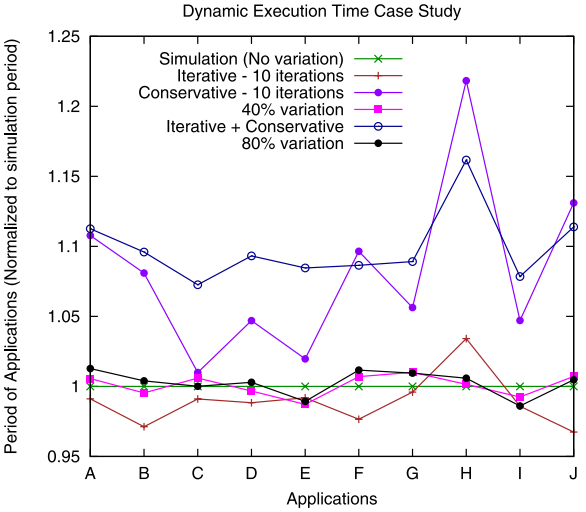


Fig. 3.19 Comparison of periods with variable execution time for all applications. A new conservative technique is applied; the conservation mechanism is used only for the last iteration after applying the base iterative analysis for 10 iterations

Varying Execution Times

Many applications are dynamic in nature (Theelen et al. 2006). When there is a variation in the execution time of the application tasks, the SDF graph is not able to capture their exact behaviour. The techniques that are conventionally used to analyze the application behaviour give an even more pessimistic bound. To evaluate the performance of our technique, we re-ran the simulation of the ten-application use-case by varying the execution time of the application tasks. Two sets of experiments were done – one by allowing the execution time to vary within 10 time units, and another within 20. The average of all task execution times was about 25. Therefore, this implied a variation of up to 40% in the first case and up to 80% in the other. Figure 3.19 shows the results of experiments when the execution time was allowed to vary in a uniform interval. A couple of observations can be made. First, the period of applications when execution time is allowed to vary does not change too much. In our experiments it varied by at most 2%. Clearly, it may be possible to construct examples in which it does vary significantly, but this behaviour was not observed in our applications. Second, the conservative analysis still gives results that are more than the period of applications with variable execution times. In this figure, we also see the difference in applying conservative analysis throughout the ten iterations, and in applying this analysis for only the last iteration. While in the former case, the prediction is sometimes very close to the measured results (application C) and sometimes very far (application H), in the latter, the results make a nice envelope that is on average 10% more than the measured results.

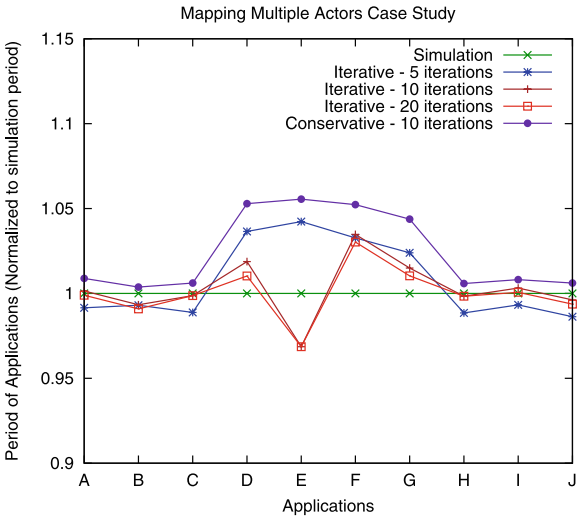


Fig. 3.20 Comparison of application periods when multiple actors of one application are mapped on one processor

Mapping Multiple Actors

So far in the above experiments we have only considered cases when one actor per application is mapped on one processor. Since each application in the experiment had up to ten actors, we needed ten processors. Clearly, this is not realistic. Therefore, we mapped multiple actors of an application on a single processor and checked whether the iterative approach still works in that case. Since we do not consider intra-task dependency, the analysis remains the same, except that there are potentially more actors on any processor causing contention. For this experiment, we assumed that we had only four processors available and the mapping is already specified at design-time. Figure 3.20 shows the comparison of the predicted results with the measured performance. The average error (mean absolute deviation) in this experiment is just 1%, while the maximum deviation is 3%. This shows that the approach is effective even when multiple actors of the same application are mapped on a resource. Further, in this experiment some processors had up to 30 actors mapped. This shows that the approach scales well with the number of actors mapped on a processor.

Mobile Phone Case Study

In this section, we present results of a case-study with real-life applications. We did not do any optimization to the application specifications and granularity obtained from the literature, in order to put our analysis to an extreme test. We consider 5 applications – video encoding (H263) (Hoes 2004), video decoding (Stuijk 2007), JPEG decoding (de Kock 2002), modem (Bhattacharyya et al. 1999), and a voice call

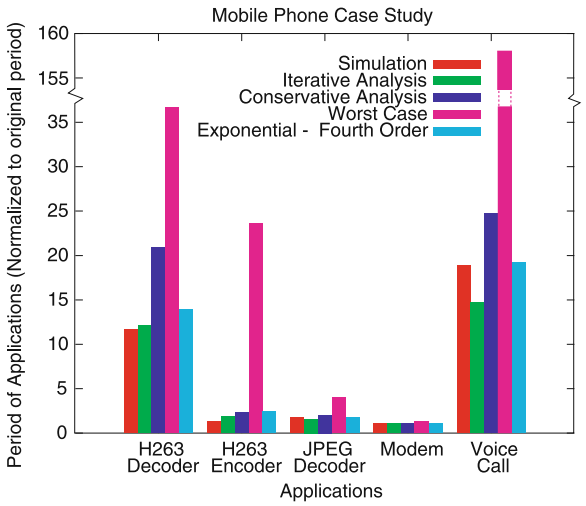


Fig. 3.21 Comparison of performance observed in simulation as compared to the prediction made using iterative analysis for real applications in a mobile phone

scenario. These applications represent a set of typical applications – often executing concurrently – on a modern mobile phone. Sufficient buffer-space is assumed to be present among all channels in the applications, such that applications do not deadlock due to lack of buffer-space. This buffer-space on each channel (just enough to avoid deadlock) and auto-concurrency of one was modeled in the application graphs to compute throughput using the SDF^3 tool.

This set of applications poses a major challenge for performance prediction since they consist of tasks with varying granularity of execution times, e.g. *anti-aliasing* module of *mp3 decoder* takes 40 time-units while the *sub-inversion* module of the same application requires 186,500 time units. Thus the assumption of the basic method that the execution times are roughly equal does not hold. Further, the repetition vectors of these applications vary significantly. While the sum of repetition vector entries of *JPEG* is 26 i.e. actors of *JPEG* have to compete for processor resources to become available 26 times for one iteration, the sum of repetition vector entries of *H263 decoder* is 1190. Further, the number of tasks in each application vary significantly. While *H263 decoder* has only four tasks, the modem application has a total of 14 tasks. For this case study, one task was mapped to one processor for each application, since multiple actor mapping options would have resulted in a huge number of potential mappings. This implied that while some processors had up to five actors, some processors only had one actor mapped. Thus, this case-study presents a big challenge for any performance prediction mechanism, and our iterative probabilistic technique was used to predict performance of these applications executing concurrently.

Figure 3.21 shows the comparison between the prediction of the iterative analysis and the simulation result. For these results, a bar chart is used instead of lines to make the graph more readable. Using a line would squeeze all the points of the *modem*, for example, to a single point. Further, it is difficult to make the gap in y-axis (needed for *voice call*) meaningful using lines. The simulation was carried out

for 100 million time units. The results are normalized with the original period of each application. The results of the bound provided by the worst-case estimate are also shown for comparison. A couple of observations can be made from the graph. First of all, the period of applications increases in different proportions. While the period of *modem* application increases by only 1.1 times, the period of *H263 decoder* increases by almost 12 times, and that of a voice call by almost 18 times. This depends on the granularity of tasks, the number of tasks a particular application is divided into, and the mapping of tasks on the multiprocessor platform. The modem application consists of about 14 tasks, but only 6 of them experience contention. The remaining 8 tasks have a dedicated processor, and therefore have no waiting time. Further, the 6 tasks that do share a processor, are only executed once per application iteration. In contrast the *inverse-quantization* module of the *H263 decoder* executes 594 times per iteration of the decoder, and has to wait for the processor to become available each time. This causes significant degradation in its performance. The second observation we can make is that the iterative analysis is still very accurate. The average deviation in throughput estimate is about 15%, and the maximum deviation is in the *voice call* application of 29%. The basic approach to fourth-order approximation also performs quite well for this case-study, and the average deviation is the same as the iterative approach, but the maximum deviation is 47%. The worst-case estimate in contrast is almost 18 times overly pessimistic. The prediction in the *voice call* application is actually 158 times of the original period. Another interesting observation is that the worst-case bound of the *modem* application is only 15% pessimistic. This is because most actors of this application do not have any contention. It should be mentioned that in this experiment first-come-first-serve arbitration was used. A different arbitration mechanism and a better mapping can distribute the resources more evenly.

Comparison with an FPGA Multiprocessor Implementation

In addition to POOSL and the analysis approaches, we also used a prototyping approach (as presented in Fig. 3.1) to test performance of multiple applications on a real hardware multiprocessor platform. (The applications presented earlier are too big to be accommodated in our FPGA multi-processor platform.) A Microblaze-based multiprocessor platform was built using the MAMPS tool (MAMPS 2009; Kumar et al. 2008). This tool generates desired architecture for Xilinx-based FPGAs using their soft-processor (Microblaze) and point-to-point connections for data transfers using Fast Simplex Links – FIFOs. Application C-code for the corresponding processors is then used and performance is measured for multiple applications. Here we present results for Sobel (edge-detection algorithm) and JPEG encoding applications.

Figure 3.22 shows the SDF model for Sobel and JPEG encoders. The Sobel model is based on pixel-level granularity while the JPEG model is based on macroblock granularity. The execution times shown in this figure are obtained by profiling the C-code of the corresponding applications on Microblaze processors, and

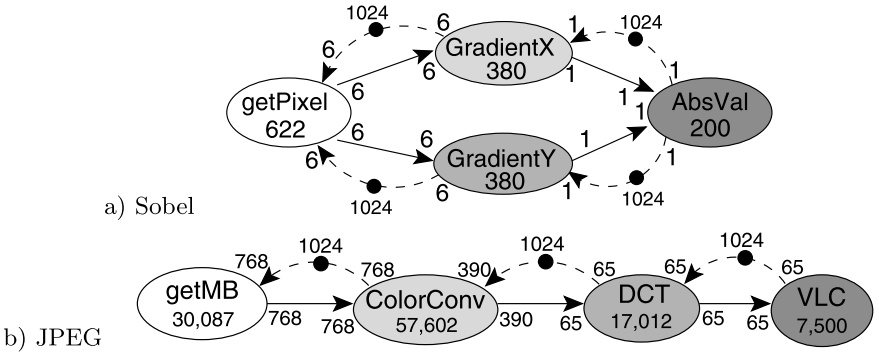


Fig. 3.22 SDF model of Sobel algorithm for one pixel, and JPEG encoder for one macroblock

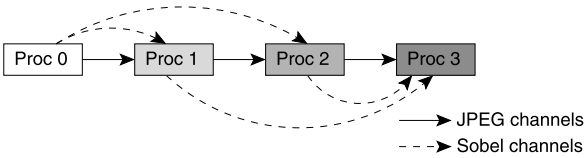


Fig. 3.23 Architecture of the generated hardware to support Sobel and JPEG encoder

Table 3.4 The period of concurrently executing Sobel and JPEG encoder applications as measured or analyzed

Application	FPGA	POOSL		Iterative P^3	
		Period	Error	Period	Error
Sobel	17,293	17,134	0.92%	16,589	4%
JPEG Enc.	103,672	104,451	0.75%	103,686	0.01%

include the communication delay for sending and receiving the data as well. As can be seen the two applications have very different granularity of actors and poses a challenge for any analysis algorithm. Figure 3.23 shows the generated hardware platform to support these two applications. The dedicated point-to-point links generated are shown by arrows. All links had a buffer-capacity of 1024 integers. This buffer limitation is modeled as a back-edge in the application graph in Fig. 3.22.

Table 3.4 shows the period of both the applications as measured on this FPGA platform. The estimates obtained by simulating the SDF models using POOSL and our proposed iterative technique are also shown. The error in these estimates in comparison with the results measured on the FPGA board are also shown. The error in estimates from POOSL is less than a percent while the maximum error in our iterative technique results is 4%.

Table 3.5 The number of clock cycles consumed on a Microblaze processor during various stages, and the percentage of error (both average and maximum) and the complexity

Algorithm/Stage	Clock cycles	Error (%) avg/max	Complexity
Load from CF Card	1,903,500	–	$O(N.n.k)$
Throughput Computation	12,688	–	$O(N.n.k)$
Worst Case	2,090	72.6/83.1	$O(m.M)$
Second Order	45,697	22.3/44.5	$O(m^2.M)$
Fourth Order	1,740,232	9.9/28.9	$O(m^4.M)$
Iterative – 1 Iteration	15,258	12.6/36	$O(m.M)$
Iterative – 1 Iteration*	27,946	12.6/36	$O(m.M + N.n.k)$
Iterative – 5 Iterations*	139,730	2.2/3.4	$O(m.M + N.n.k)$
Iterative – 10 Iterations*	279,460	1.9/3.0	$O(m.M + N.n.k)$

N – number of applications, n – number of actors in an application, k – number of throughput equations for an application, m – number of actors mapped on a processor, M – number of processors

*Including throughput computation time

Implementation Results on an Embedded Processor

The proposed algorithms were ported to an embedded processor – Microblaze; this required some rewriting to optimize the implementation for timing and reduced memory use. The default time taken for second and fourth order, for example, was 72M and 11M cycles respectively. Table 3.5 shows the time taken during various stages and algorithms after rewriting. The algorithmic complexity of each stage and the error as compared to the simulation result is also shown.

The error in various techniques as compared to the performance achieved is also shown in Table 3.5. As can be seen, the basic probability analysis with fourth order gives an average error of about 10% and a maximum error of 29%. The iterative technique after just five iterations predicts a performance that is within 2% of the measured performance on average and has only 3% maximum deviation in the entire set of applications.

This system consists of the same 10 applications as used in the previous subsection. The loading of application properties from the CF card took the most amount of time. However, this is only done once at the start of the system, and hence does not cause any bottleneck. On our system operating at 50 MHz, it takes about 40 ms to load the applications-specification. Parametric throughput computation is quite fast, and takes about 12K cycles for all 10 applications. (It should be mentioned that here we merely need to evaluate the throughput expressions that are computed for each application. Derivation of expressions is still time consuming and not done at run-time.) The probabilistic analysis itself for all the applications is quite fast, except the fourth-order analysis.

For the iterative analysis, each iteration takes only 15K cycles i.e. 300 microseconds. If 5 iterations are carried out, it takes a total of 140K cycles for all 10 applications including the time spent in computing throughput. This translates to a time of about 3 ms on 50 MHz processor when the performance of all ten applications is computed. Since starting a new application is likely to be done only once in every few seconds or minutes, this is a small overhead. Further, the processor speed of a typical processor in an embedded device is around 400–500 MHz. Thus, it will take only about 300 microseconds.

4 Suggested Readings

In (Bambha et al. 2002), the authors propose to analyze the performance of a *single application* modeled as an SDF graph by decomposing it into a homogeneous SDF graph (HSDFG) (Sriram and Bhattacharyya 2000). The throughput is calculated based on analysis of each cycle in the resulting HSDFG (Dasdan 2004). However, this can result in an exponential number of vertices (Pino and Lee 1995). Thus, algorithms that have a polynomial complexity for HSDFGs therefore have an exponential complexity for SDFGs. Algorithms have been proposed to reduce average case execution time (Ghamarian et al. 2006), but it still takes $O(n^2)$ in practice where n is the number of vertices in the graph. When mapping needs to be considered, extra edges can be added to model resource dependencies such that a complete analysis taking resource dependencies into account is possible. However, the number of ways this can be done even for a single application is exponential in the number of vertices (Kumar et al. 2006b); for multiple applications the number of possibilities is infinite. Further, only static order arbitration can be modeled using this technique.

For *multiple applications*, an approach that models resource contention by computing *worst-case-response-time* (WCRT) for TDMA scheduling (requires preemption) has been analyzed in (Bekooij et al. 2005). This analysis gives a very conservative bound. Further, this approach requires preemption for analysis. A similar worst-case analysis approach for round-robin is presented in (Hoes 2004), which also considers non-preemptive systems, but suffers from the same problem of lack of scalability. WCRT is computed by adding the execution times of all the actors mapped on a resource. Thus, the response time of an actor a_j is given by:

$$t_{resp}(a_j) = \sum_{i=1}^{N} t_{exec}(a_i) \quad \forall j = 1, 2, \ldots N \tag{3.25}$$

However, as the number of applications increases, the bound increases much more than the average case performance. Real-time calculus has also been used to provide worst-case bounds for multiple applications (Richter et al. 2003; Thiele et al. 2000; Kunzli et al. 2006; Schliecker and Ernst 2009; Leontyev et al. 2009). Besides providing a very pessimistic bound, the analysis is also very intensive and requires a very large design-time effort. On the other hand our approach is very simple. However, we should note that above approaches give a worst-case bound that is targeted at hard-real-time (RT) systems.

A common way to use probabilities for modeling dynamism in application is using stochastic task execution times (Abeni and Buttazzo 1999; Manolache et al. 2004; Hua et al. 2007). In our case, however, we use probabilities to model the resource contention and provide estimates for the throughput of applications. This approach is orthogonal to the approach of using stochastic task execution times. In our approach we assume fixed execution time, though it is easy to extend this to varying task execution times as well, as shown by the results. To the best of our knowledge, there is no efficient approach of analyzing multiple applications on a non-preemptive heterogeneous multi-processor platform.

Queuing theory also allows computing the waiting times when several processes are being served by a resource (Takacs 1962) and has been applied for networks (Robertazzi 2000) and processor-sharing (Coffman et al. 1970). However, this is not applicable in our scenario for a number of reasons. First, since we have circular dependencies in the SDF graphs, feedback loops are created that cannot be handled by the queuing theory. Secondly, the execution time of tasks on a processor does not follow a common distribution. Each task may have an independent execution time distribution. Therefore, a general expression for the service time for tasks mapped on a processor cannot be determined.

5 Conclusions

In this chapter, we saw a probabilistic technique to estimate the performance of applications when sharing resources. An iterative analysis is presented that can predict the performance of applications very accurately. Besides, a conservative flavour of the iterative analysis is presented that can also provide conservative prediction for applications for which the mis-prediction penalty may be high.

The basic probability analysis gives results with an average error of 10%, and a maximum error is 29%. In contrast, the average error in prediction using iterative probability is only 2% and the maximum error of 3%. Further, it takes about four to six iterations for the prediction to converge. The complexity and execution time of the algorithm is very low – it takes only 3 ms to evaluate the performance of ten applications on a 50 MHz embedded processor.

Further, we presented results of a case-study of applications commonly used in a mobile phone. The models of these applications vary in the number of tasks, granularity of tasks, and also the repetition vectors of the applications. The simulation result of executing all applications concurrently is compared with the iterative analysis. Even in this particular use-case, the prediction by iterative analysis is close to the simulation result. This proves the robustness of the technique. We also see that applications with coarser task granularity perform better in the first-come-first-serve arbitration as compared to applications that have a finer granularity. This occurs since the tasks with finer granularity have to compete for resources more often. Different arbitration mechanisms can potentially alleviate this problem, and more research should be done into that.

Chapter 4
Resource Management

Modern multimedia systems consist of a number of resources. This includes not just computational resources, but also others, for example, memory, communication bandwidth, energy. Resources have to be allocated to applications executing on the system such that all applications can produce the expected result. When dealing with multimedia embedded systems, ensuring that all applications can meet their *non-functional* requirements, e.g. throughput, also becomes important. As has been emphasized in the earlier chapters, this poses a significant challenge when multiple applications are executing. Further, when run-time admission *and* addition of applications is supported, e.g. in modern mobile phones, this challenge takes yet another dimension. A complete analysis is infeasible at design-time due to two major reasons: (1) little may be known at design-time about the applications that need to be used in future, and (2) the number of design-points that need to be evaluated is prohibitively large. A run-time approach can benefit from the fact that the exact application mix is known, but the analysis has to be fast enough to make it feasible. Resource management – simply said – is managing the resources available on the multiprocessor platform.

In this chapter, we present a design-flow for designing systems with multiple applications. We present a hybrid approach where the time-consuming application-specific computations are done at design-time, and isolated from other applications – to maintain composability, and the use-case-specific computations are performed at run-time. Further, we present a resource manager (RM) for the run-time aspects of resource management. The need for a middle-ware or OS-like component for the MPSoC has already been highlighted in literature (Wolf 2004). We highlight two main uses of the RM – *admission control* and *budget enforcement*, that are essential to ensure that the performance requirements of all applications are met.

The remainder of this chapter is organized as follows. Section 1 explains the properties that are extracted from individual applications off-line. Section 2 explains how these properties are used by the resource manager at run-time to perform admission control and enforcing budgets. Section 3 provides an approach for achieving predictability through suspension. The results of a case study done with an H263 and a JPEG decoder are discussed in Sect. 4. Section 5 provides suggested readings for

A. Kumar et al., *Multimedia Multiprocessor Systems*, Embedded Systems,
DOI 10.1007/978-94-007-0083-3_4,

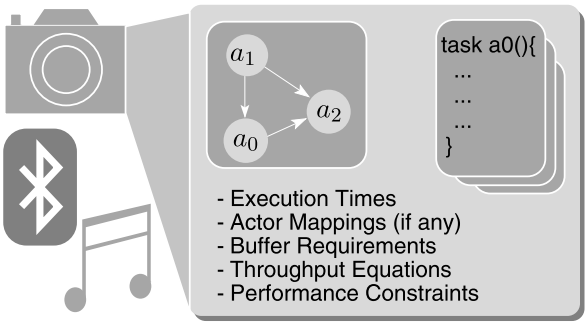


Fig. 4.1 Off-line application(s) partitioning, and computation of application(s) properties. Three applications – photo taking, bluetooth and music playing, are shown above. The partitioning and property derivation is done for all of them, as shown for photo taking application, for example

research that has been done, and Sect. 6 ends the chapter with some conclusions and directions for future work.

1 Off-line Derivation of Properties

As has been explained earlier, it is often not feasible to know the complete set of applications that the system will execute. Even in cases, when the set of applications is known at design-time, the number of potential use-cases (or scenarios) may be large. The combination of off-line and on-line (same as run-time) processing keeps the design-effort limited. Note that off-line is different from design-time; while system design-time is limited to the time until the system is rolled-out, off-line can also overlap with using the system. In a mobile phone for example, even after a consumer has already bought the mobile phone, he/she can download the applications whose properties may be derived after the phone was already designed. In our methodology, some applications may be unknown even at design-time. In those cases, the properties of the applications are derived off-line, and the run-time manager checks whether the given application-mix is feasible.

Figure 4.1 shows the properties that are derived from the application(s) off-line. Individual applications are partitioned into tasks with respective program code tagged to each task and communication between them explicitly specified. The program code can be profiled (or statically analyzed) to obtain execution time estimates for the actors. For this chapter, we assume that the application is already split into tasks with worst-case execution-time estimates. A complete survey of the methods and tools available for computing worst-case execution-times is provided in (Wilhelm et al. 2008).

The following information is extracted from the application off-line as shown in Fig. 4.1.

- Partitioned program code into tasks.
- SDF model of the application.
- Worst-case execution time estimates of each task.
- Minimum performance (throughput) needed for satisfactory user-experience.
- Mapping of these tasks on to the heterogeneous platform.

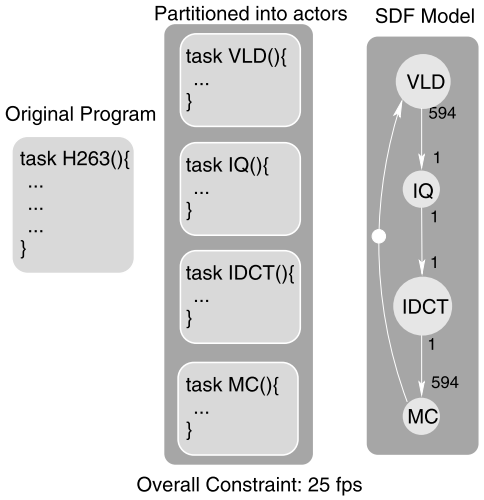


Fig. 4.2 The properties of H263 decoder application computed off-line

$$T_1 = 0 \times t_{vld} + 593 \times t_{iq} + 594 \times t_{idct} + 1 \times t_{mc}$$
$$T_2 = 1 \times t_{vld} + 594 \times t_{iq} + 593 \times t_{idct} + 0 \times t_{mc}$$

Task	Mapping	Execution cycles	Min outgoing buffer
VLD	ARM7	26018	594 tokens
IQ	ARM9	559	1 tokens
IDCT	TIC67	486	594 tokens
MC	TIC64	10958	–

- Buffer-sizes needed for the edges in the graph.
- Throughput equations of the SDF model.

Note that there may be multiple *Pareto-points* with different mappings, buffer-sizes and throughput equations. Figure 4.2, for example, shows how the application partitioning and analysis is done for H263 decoder application. The SDF model presented in this figure has been taken from (Stuijk 2007). Note that the strong connectedness in the graph comes from the back-edges corresponding to the buffer between the actors. These are omitted in the graph for clarity. The sequential application code is split into task-level descriptions, and an SDF model is derived for these communicating tasks. The corresponding production and consumption rates are also mentioned along the edges. The table alongside the figure shows the mapping and worst case execution times of each task. (The original model of (Stuijk 2007) is based on ARM7 implementation; the actual cycle-counts on other processors may vary.) The buffer-size needed between each actor is also mentioned in the table. There are two throughput expressions that correspond to this buffer-size (Ghamarian et al. 2008), where t_a shows the response time of the actor a. These expressions limit the maximum throughput for this particular model of H263 (under these buffer-size constraints). In order to compute the actual period, both T_1 and T_2 are evaluated for the particular response-time combination of these actors. The larger of the two gives the period of H263 decoder. For these initial execution time

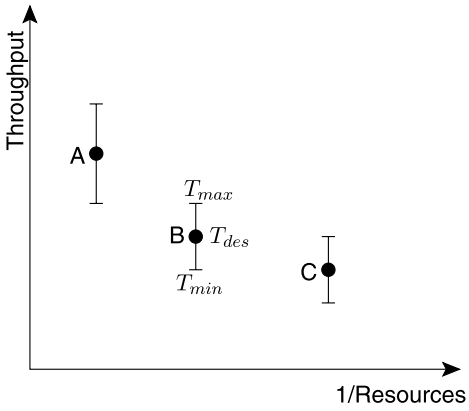


Fig. 4.3 Boundary specification for non-buffer critical applications

estimates, the first expression forms the bottleneck and limits the period to 646,262 cycles. This implies that if each of these tasks is executed on a processor of 50 MHz, the maximum throughput of the application is 77 iterations per second. (In practice, the frequency of different processors may be different. In that case, we should add the time taken for each task in throughput expressions, instead of cycles.) Clearly, when this application is executing concurrently with other applications, it may not achieve the desired throughput. For this example, we have assumed that the minimum performance associated with this application is 25 frames per second. This is the constraint that should be respected when the application is executed.

Performance Specification

An application can often be associated with multiple quality levels as has been explained in existing literature (Nollet 2008; Lauwereins et al. 2002). In that case, each quality of the application can be depicted with a different task graph with (potentially) different requirements of resources and different performance constraints. For example, a bluetooth application may be able to run at a higher or lower data rate depending on the availability of the resources. If a bluetooth device wants to connect to a mobile phone which is already running a lot of jobs in parallel, it may not be able to start at 3.0 Mbps (Bluetooth 2.0 specification (Bluetooth 2004)) due to degraded performance of existing applications, but only at 1.0 Mbps (Bluetooth 1.2 specification (Bluetooth 2004)). Figure 4.3 shows an example of how performance bounds are specified in the form of Pareto-points. The figure shows three quality levels – A, B and C. Each quality level has different throughput constraints and require different amounts of resources. Note that along x-axis, the amount of resources decreases as we move to the right. In the figure, the middle point T_{des} indicates the desired throughput, and T_{max} and T_{min} indicate the maximum and minimum throughput that an application may tolerate respectively.

However, this only applies to applications that do not have an input or output throughput constraint. For applications where jitter is more critical than the average

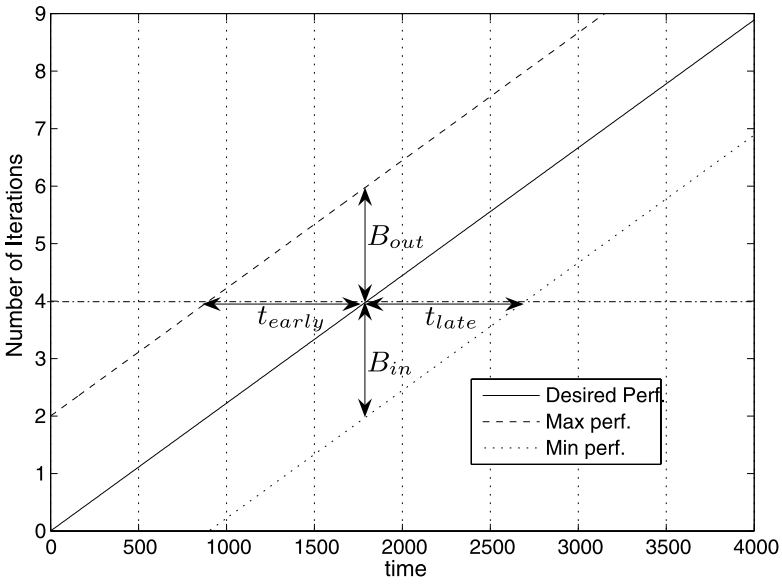


Fig. 4.4 Boundary specification for buffer-critical applications or constrained by input/output rate

throughput, for each Pareto point in Fig. 4.3, the performance bound should be as specified in Fig. 4.4. The deviation from the desired throughput in the vertical direction indicates that extra buffer space is needed. The same is indicated in Fig. 4.4 by B_{out} and B_{in} to indicate the maximum output and input buffer present for the application respectively. If an application is running two iterations ahead of the desired rate, the output of those two iterations needs to be buffered somewhere. The same applies to the input buffer as well when an application is lagging behind. The deviation in the horizontal direction signifies how *early* or *late* the application completes its execution as compared to the desired time. It should be noted that the long-term average throughput i.e. the number of iterations per second, is the same for all the three lines.

2 On-line Resource Manager

Since the entire analysis cannot be performed at design-time, we also need a resource manager (RM) on-line. It controls the access to resources – both critical and non-critical, and enforces their usage. Typically it takes care of resource assignment, budget assignment and enforcement, and admission control. For example, when an actor can be mapped on multiple processors, or when multiple instances of one processor are available, it chooses which one to assign to the actor. It also assigns and enforces budgets, for example, on shared communication resources like a bus or on-chip network e.g. Æthereal (Goossens et al. 2005). All the tasks of RM can be

categorized in two functions – *admission control* and *budget enforcement*. When a new job arrives in the system and needs resources, the resource manager checks the current state of the system and decides whether it has enough resources to accommodate it, and at which quality and resource-binding. It also enforces a specified resource budget for a task to ensure it only uses what was requested. These two functions are explained below in more detail.

Admission Control

One of the main uses of a resource manager is admitting new applications. In a typical high-end multimedia system, applications are started at run-time, and determining whether there are enough resources in the system to admit new applications is non-trivial. When the RM receives a request for a new application to be started (through the user interface, or through another application already running in the system), it fetches the description of the incoming application from the memory. The description contains the following information:

- *Performance specification:* specification of the desired throughput together with the bounds.
- *Actor array:* A list of actors along with their execution times, repetition vector and a list of nodes to which they can be mapped to.
- *Throughput equations:* The equations that provide the limiting expressions to evaluate throughput during contention.

It should be mentioned that when network and memory resources are also considered, the structure of the application graph, memory and communication requirements of the graph also need to be provided. The above information can also be in form of a *Pareto-curve* where each point in the curve corresponds to a desired quality of the application and specification as above. With this information, the admission controller checks whether there are enough resources available for all the applications to achieve their desired performance using a *predictor* (or in the case of a Pareto-curve, at which point the application can be run). The predictor estimates the performance of applications for the desired quality level and mapping, as explained below.

Performance Predictor

The performance predictor runs as part of admission-controller and uses the off-line information of the applications to predict their performance at run-time. For example, imagine a scenario where you are in the middle of a phone call with your friend and you are streaming some mp3 music via the 3G connection to your friend, and at the same time synchronizing your calendar with the PC using bluetooth. If you also wish to now take a picture of your surrounding, traditional systems will

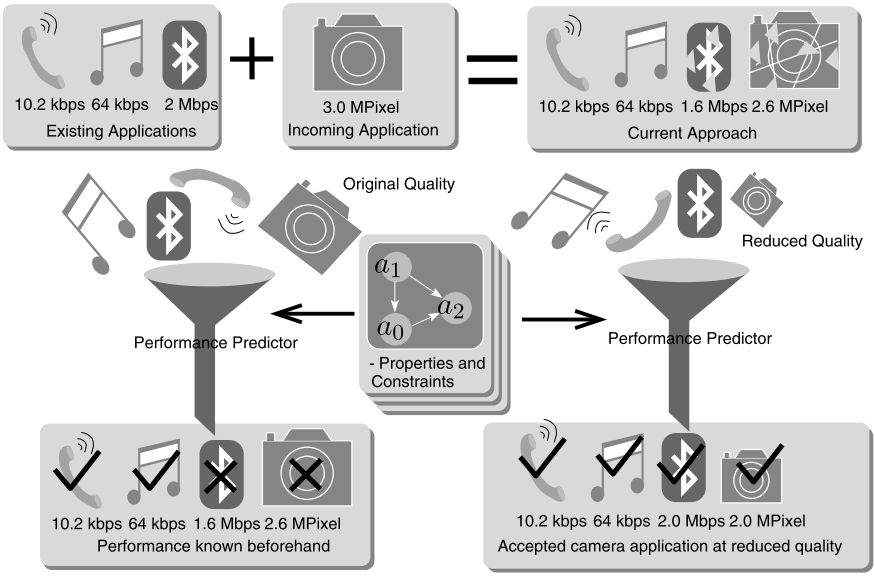


Fig. 4.5 On-line predictor for multiple application(s) performance

simply start the application without considering whether there are enough resources to meet the requirements or not. As shown in Fig. 4.5, with so many applications in the system executing concurrently, it is very likely that the performance of the camera and the bluetooth application may not be able to match their requirements.

The predictor uses the iterative probabilistic mechanism, as explained in Chap. 3. With this mechanism, using the properties of applications computed off-line, we can compute the expected performance before admitting the application. It can then be decided to either drop the incoming application, or perhaps try the incoming application (or one of the existing applications, if allowed) at a lower quality level or with a different mapping. As shown in Fig. 4.5, if the camera application is tested at 2.0 MPixel requirements, all the applications can meet their requirements. It is much better to know in advance and take some corrective measure, or simply warn the user that the system will not be able to cope up with these set of applications, and suggest an alternative.

Resource Assignment

When an actor can be mapped on multiple resources (either because it can be mapped on different types of processors, or because there are multiple instances of the type of processors it can be mapped on, or both), the admission controller iterates over all options until a suitable option is found. Heuristics to explore mapping options efficiently are orthogonal to our approach. One example to make this exploration more efficient is to look at processor utilization results from the iterative

probability mechanism and shift tasks from processors with high utilization to those with low utilization. Such heuristics can be used in combination with our approach.

Task Migration

If task migration is considered, there are a lot more design-points to explore. However, since we have a fast analysis mechanism, with a fast embedded processor it may be feasible. It should be mentioned that supporting task migration at the architectural level is also difficult. It has to be ensured that the program code of the affected actor can be seamlessly transferred to the new processor. Further, the network connections whose sink or source is the affected actor, have to be torn down and set-up again. During this transition, care has to be taken that no data is lost in the affected buffers. However, having said that, there is already quite some work done in this area. A technique to achieve low-cost task migration in heterogeneous MPSoC is proposed in (Nollet et al. 2005). Yet another approach is presented in (Bertozzi et al. 2006). In our design flow, we do not support task-migration, though the design-flow and the analysis techniques allow for it.

It can be seen how this flow allows addition of applications at run-time without sacrificing predictability. The user can download new applications as long as the application is analyzed off-line and the properties mentioned earlier are derived. Since the particular use-case performance analysis is done at run-time, no extensive testing is needed at design-time to verify which applications will meet their performance requirements and which not. When the performance of all applications is found to be satisfactory, then the resource manager *boots* the application. This translates to loading the application code from memory (possibly external) into local memories of respective processors and enabling/adding the application in the processor arbiters. Once the application is started, it sends a signal to the RM at the end of every iteration to keep it updated.

Resource Budget Enforcement

This is another important function for the resource manager. When multiple applications are executing in a system (often with dynamically varying execution times), it is quite a challenge to schedule all of them such that they meet their throughput constraints. Using a static scheduling approach is neither scalable, nor adaptive to dynamism present in a system (Kumar et al. 2006b). A resource manager, on the other hand, can monitor the progress of applications running in the system and enforce the budgets requested at run-time.

Clearly, the monitoring and enforcement also has an overhead associated with it. Granularity of control is therefore a very important consideration when designing the system, and determines how often the RM inspects the system. We would like to have as little control as possible while achieving close to desired performance. In our approach, the budgets are enforced at the application-level to limit the overhead.

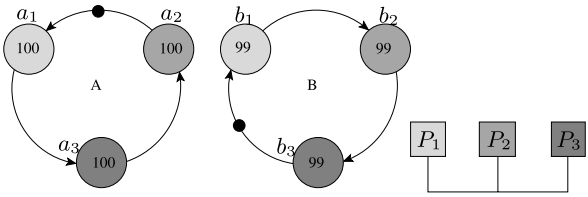


Fig. 4.6 Two applications running on same platform and sharing resources

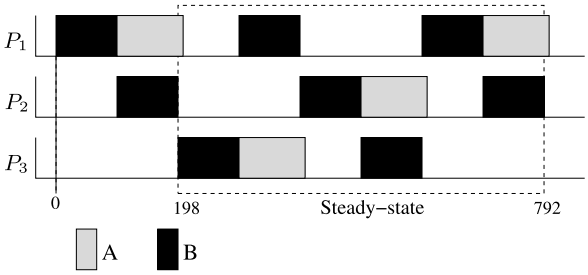


Fig. 4.7 Schedule of applications in Fig. 4.6 running together. The desired throughput is 450 cycles per iteration

Motivating Example

Two SDF graphs (similar to the ones in Fig. 2.13) are shown in Fig. 4.6. Each graph has three actors scheduled to run on three processors as indicated by the shading. The flows of applications *A* and *B* are reversed with respect to each other to create more resource contention. The execution time of each actor of *B* is reduced to 99 clock cycles to create a situation where *A* experiences a worst-case waiting time. This can also correspond to a situation in which all actors of both *A* and *B* have an execution time of 100 clock cycles, but actors of *B* finish just before actors of *A*, or that actors of *B* are checked first. Let us further assume that the desired maximum period for both applications is 450 clock cycles.

If each application is running in isolation, the achieved period would be 300 and 297 time units for *A* and *B* respectively. Clearly, when the two are running together on the system, due to contention they may not achieve this performance. In fact, since every actor of *B* always finishes just before *A*, they always get the desired resource and *A* is not able to achieve the required performance; while *B* on the other hand achieves a better performance than necessary. The corresponding schedule is shown in Fig. 4.7.

If the applications are allowed to run without intervention from the RM, we observe that it is not possible to meet the performance requirements; the resulting schedule for executing *A* and *B*, which is highly unpredictable at compile time, yields a throughput of *B* that is twice the throughput of *A*. However, if *B* could be temporarily suspended, *A* will be able to achieve the required throughput. A resource manager can easily provide such a control and ensure that desired throughput for both *A* and *B* is obtained.

We also see in the example that even though each application only uses a third of each processing node, thereby placing a total demand of two-third on each processing node, the applications are not able to achieve their required performance.

A compile-time analysis of all possible use-cases can alleviate this problem by deriving how applications would affect each other at run-time. However, the potentially large number of use-cases in a real system makes such analysis infeasible. A resource manager can shift the burden of compile-time analysis to run-time monitoring and intervention when necessary.

Suspending Applications

We now discuss how this suspension is implemented. Each application sends a signal to the RM upon completion of each iteration. This is achieved by appending a small message at the end of the output actor of the application graph, i.e. the last task in the execution of an application after whose execution an iteration of the application can be said to have been completed. (In the event of multiple output actors, any output actor may be chosen for this.) This allows the RM to monitor the progress of each application at little cost. After every sample period – defined as *sample points*, the RM checks whether all applications are progressing at their desired level. If any application is found to be running below the desired throughput, the application which has the most slack (i.e. the highest ratio of achieved to desired throughput) is suspended. The suspended application is re-activated when all applications are running above the desired throughput. Suspension and re-activation occur only at sample points.

Each arbiter maintains two lists – an actor ready queue, and an application enable list. Once the processor is available, the arbiter checks the actor at the head of the queue, and if its corresponding application is enabled, it is executed. Otherwise, the next available actor (with enabled application) is executed. Suspension of an application is achieved by sending a temporary disable signal to the arbiters running the application. Say, for example, if application A has actors mapped on 3 processors P_1, P_2 and P_3, then the three processor-arbiters will be signalled to disable application A. Thus, even when actors of application A are ready, they will not be executed.

Suspension of an application is not to be confused with preemption. In our approach, we do not allow actors to be preempted in the middle of their execution; actors execute atomically. However, an application can be suspended after completing the execution of any actor. This limits the context that needs to be saved when an actor is in the middle of its execution.

Suspension Example

Figure 4.8 shows an example interaction diagram between various modules in the design. The user-interface module sends a request to start application X (1). The resource manager checks if there are enough resources for it, and then admits it in the system (2). Applications Y and Z are also started respectively soon after as indicated on the figure (3–6). However, when Z is admitted, Y starts to deteriorate in

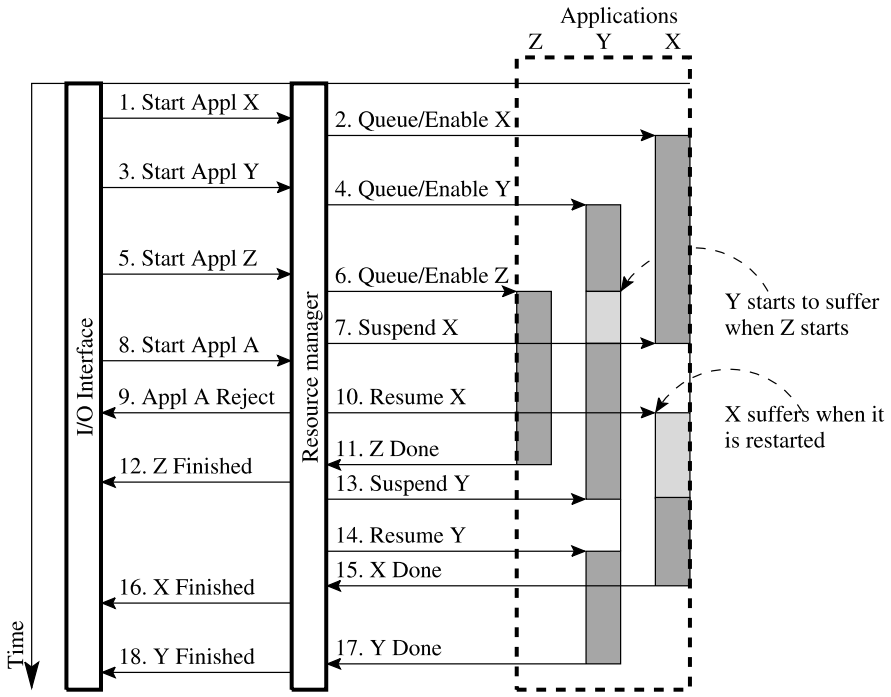


Fig. 4.8 Interaction diagram between user interface, resource manager, and applications in the system-setup

performance. The resource manager then sends a signal to suspend X (7) to the platform because it has slack and Y is then able to meet its desired performance. When X resumes (10), it is not able to meet its performance and Y is suspended (13) because now Y has slack. When the applications are finished, the result is transmitted back to the user-interface (12, 16 and 18). We also see an example of application A being rejected (9) since the system is possibly too busy, and A is of lower priority than applications X, Y and Z.

Communication Overhead

The overhead of monitoring and suspension is easy to compute. Consider ten applications running concurrently on a ten-processor system. Each application signals the RM every time it completes an iteration. Let us assume the period of each application is 100,000 clock cycles. Therefore, on average only one message is sent to the resource manager every 10,000 cycles. Let us consider the case with sampling period being 500,000 cycles. The RM sends messages at most every processing node every sampling interval. This is on average, one message every 50,000 cycles, giving in total 6 messages every 50,000 cycles – 5 from the application to the resource manager and 1 from the resource manager to the processor. If the length of

each message is around 10 bytes, we get a bandwidth requirement of 60 bytes every 50,000 cycles. For a system operating at 100 MHz, this translates to about 120 kilobytes per second. In general, for N applications $A_0 \ldots A_{N-1}$ each with throughput T_{A_i} mapped on M processing nodes, with the RM sampling at f_{samp} frequency, if each message is b bytes, the total communication bandwidth is given by following equation

$$BW_{reqd} = b \times \left(f_{samp}.M + \sum_{i=0}^{N-1} T_{A_i} \right) \tag{4.1}$$

This is only the worst-case estimate. In practice, messages from the RM will not be sent to every processing node every sampling interval, but only when some application is suspended or resumed.

The communication can be further reduced by sending the suspension and enabling message to only one of the nodes (perhaps output or input actor) of the application. Suspending any actor of an application that is strongly connected will eventually suspend the whole application, since there are buffer dependencies in any system. For example, if the output actor is disabled, the data which this actor would consume will not be used and the producing actor for that data will not be able to execute any more. This is also known as *back-pressure*, and will eventually stall the entire application. Similarly, when the output actor is enabled, it will empty the buffers and gradually all the actors will start executing. However, this implies that it will take more time for the entire application to be suspended and enabled, unlike in the case when all actors are suspended simultaneously.

Arbiter vs Resource Manager

We have seen in the earlier sections how a resource manager helps all applications achieve their desired performance by suspending some applications. In some sense, it is similar to the processor arbiter. However, there are two key differences between the resource manager and the processor arbiter: (1) the level of control is very different, and (2) the granularity of their operation differs significantly. Figure 4.9 shows how the resource manager differs from the application QoS manager and from the processor arbiter. An application quality-of-service (QoS) manager defines the quality at which the application should be run. In multimedia applications, the quality of applications may need to be changed during execution, and this QoS manager might dictate varying desired levels of application quality at different points in time. For example, if an application goes into background, its quality level might change. Such change is unlikely to happen more than once every few seconds. On the other end of the control spectrum we have a processor arbiter, which determines the order of actors to execute on a processor. This is generally in the order of micro-seconds, but depends on the granularity of actors. The resource manager operates in between the two extremes and tries to achieve the quality specified by QoS manager by reconfiguring the arbiter once in a while.

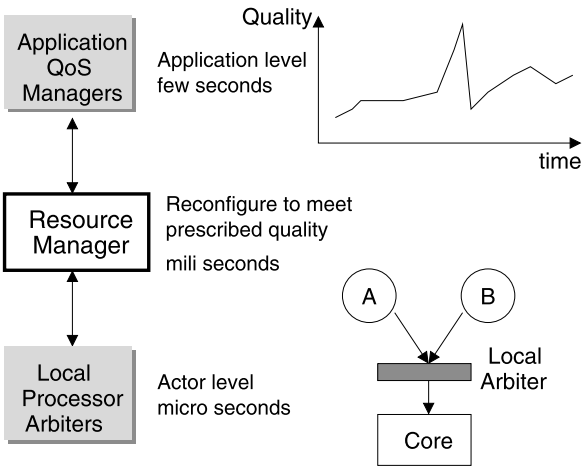


Fig. 4.9 Resource manager achieves the specified quality without interfering at the actor level

An example an arbiter is a rate-controlled-static-priority (RCSP) mechanism that may be applied at each local arbiter to achieve desired rate of actor execution at individual processors (Zhang and Ferrari 1993). However, the RCSP mechanism is not capable of handling dynamic applications. Further, an application consists of multiple actors, which can often operate at different rates in different iterations. When the rate-control is achieved via local arbiters at cores, each arbiter has to be aware of the global state of the application. This is much harder to achieve in practice, and also expensive in terms of communication. In our approach, the arbiter at each processor is very simple, and the desired rate is obtained by suspension through the resource manager.

3 Achieving Predictability Through Suspension

The previous section explains how performance of applications can be regulated by a simple suspension mechanism. However, it is not possible to provide guarantees for application performance. In this section, we show how predictability can be achieved when applications are suspended. Suspension of applications leads to different combinations of active applications, and thus affects the throughput of applications. Using techniques presented in Chap. 3, we can predict the performance in any combination of applications very accurately. The overall performance of applications can be obtained by using the weighted average of performance in individual combinations. The accuracy of the prediction depends on the accuracy of the performance prediction in individual combination.

With two applications in the system, say A_0 and A_1, there are four possibilities – none of the applications, only A_0, only A_1, and both A_0 and A_1 executing. (In general, for N applications, there are 2^N possibilities.) Let us denote them by C_j

Table 4.1 Table showing how predictability can be achieved using budget enforcement. Note how the throughput changes by varying the ratio of time in different combinations

Combination	C0	C1	C2	C3	Total
A0 throughput [Iterations/sec]	0	8	0	6	
A1 throughput [Iterations/sec]	0	0	6	4	
Time distribution 1	0	0.25	0.25	0.5	1.0
A0 effective [Iterations/sec]	0	2	0	3	5
A1 effective [Iterations/sec]		0	1.5	2	3.5
Time distribution 2	0	0	0.25	0.75	1.0
A0 effective [Iterations/sec]	0	0	0	4.5	4.5
A1 effective [Iterations/sec]		0	1.5	3	4.5

for $j = 0, 1, 2, 3$ respectively. Let us further denote the performance of application A_i in combination C_j, by $Perf_{ij}$, where performance is determined by number of iterations per cycle. Clearly $Perf_{02} = 0$, since A_0 is not active in C_2. Let us also define the ratio of time spent in combination C_j, by t_j. Clearly, $\sum_{j=0}^{3} t_j = 1$. The overall performance of the application A_i is then given by $Perf_i = \sum_{j=0}^{3} Perf_{ij}.t_j$.

Table 4.1 shows an example with two applications, A_0 and A_1. Respective performances are shown for each combination of applications. We can also see how the overall performance of each application can be controlled by different time distributions. While in the first distribution, the overall throughput of A_0 and A_1 is 5 and 3.5 iterations per second respectively, in the other it is 4.5 iterations per second for both applications.

Let us say, the required overall performance of applications is denoted by $Perf\ Reqd_i$ for Application A_i. The problem then is to determine the time-weights t_j in such a way that the required performance of all applications are met. For two applications, for example, the following equations hold:

$$t_0 + t_1 + t_2 + t_3 = 1 \tag{4.2}$$

$$t_1.Perf_{01} + t_3.Perf_{03} \geq Perf\ Reqd_0 \tag{4.3}$$

$$t_2.Perf_{12} + t_3.Perf_{13} \geq Perf\ Reqd_1 \tag{4.4}$$

(Note that strict equality may be required for some applications. For many applications, it may be sufficient to specify a minimum performance required.)

The above can be formulated as a linear programming problem, where the objective function is to minimize the total time spent in active states; or in other words, maximize the time in state C_0, i.e. t_0. We propose the following linear programming formulation.

Objective function: Minimize $\sum_{j=1}^{3} t_j$, subject to the constraints

$$\sum_{j=1}^{3} Perf_{ij}.t_j \geq Perf\,Reqd_i \quad \forall i \tag{4.5}$$

$$t_j \geq 0 \quad \forall j \tag{4.6}$$

The feasibility can be determined by checking if $\sum_{j=1}^{3} t_j \leq 1$ holds.

Reducing Complexity

As the number of applications increases the number of potential states also increases. For 10 applications, there are 1024 potential states. In those situations, solving the above linear programming problem is slowed down significantly, since its complexity is polynomial in the number of combinations (Cormen et al. 2001; Wikipedia 2008). This problem can be tackled by limiting the number of applications that are allowed to be suspended at any point in time. This reduces the number of states dramatically. For example, in the scenario with 10 concurrently active applications, if only 1 application is allowed to be suspended, we get a total of 11 combinations – 1 with all applications executing and 10 with one application suspended in each.

Limiting the number of combinations has also other advantages. At run-time the resource manager has to make a switch between all the states as and when needed. Having more states requires more memory and more switches (suspensions) to be made at run-time. This also leads to more communication traffic. In view of the run-time problems, another objective function to optimize is to minimize the total number of states. Its formulation as an optimization problem is provided below.

Objective function: Minimize $\sum_{j=1}^{3} x_j$, where $x_j = 1$ denotes that state C_j will be used at run-time, and $x_j = 0$ denotes that C_j is not used at run-time. Following constraints apply:

$$\sum_{j=0}^{3} Perf_{ij}.t_j.x_j \geq Perf\,Reqd_i \quad \forall i \tag{4.7}$$

$$t_j \geq 0 \quad \forall j \tag{4.8}$$

$$\sum_{j=0}^{3} t_j \leq 1 \tag{4.9}$$

$$x_j \in \{0, 1\} \quad \forall j \tag{4.10}$$

Unfortunately it is not possible to formulate the above as a linear programming problem; the relation between x_i and t_i is not linear. The above is an example of a mathematical programming problem. Dynamic programming can be used to solve the above problem but since it cannot be solved in linear polynomial time, if the number of states is large, it takes a long time to find a solution.

Dynamism vs Predictability

So far we have explained two different approaches for ensuring that applications can achieve their performance at run-time. The run-time approach of checking the performance of all applications and disabling applications if needed, at each *sample point*, is clearly more capable of handling dynamism in the system, while the static approach of determining the time that should be spent in different states at design-time is more predictable. With static approach one can know at design time what the run-time performance will be. However, this comes at a high memory and design-time cost. The analysis has to be done for each use-case to compute the fractions of time the system should spend in different states at run-time, and for each use-case this distribution has to be stored. As has already been mentioned earlier, as the number of applications increases in a system, so does the number of use-cases. Therefore, the static approach may lead to an explosion in the data that needs to be stored. Needless to say that when an application is added in the system at run-time, the entire linear programming analysis needs to be repeated. The dynamic approach is more adaptable since there is no design-time analysis needed, and run-time addition of applications can be easily handled.

4 Experiments

A three-phase prototype tool-flow was developed to automate the analysis of application graphs. The first phase concerns specifying different applications (as SDF graphs), the processors of the MPSoC platform (including their scheduler type), and the mapping of actors to nodes. For each application, desired throughput is specified together with the starting time of the application. After organizing the information in an XML specification for all three parts, a POOSL model (Theelen et al. 2007) of the complete MPSoC system is generated automatically. The second phase relies on the POOSL simulator, which obtains performance estimates, like the application throughput and processor utilization. It also allows generation of trace files that are used in the final phase to generate schedule diagrams and graphs like those presented in this section.

DSE Case Study

This section presents results of a case study regarding the mapping of H263 and JPEG decoder SDF models (described in (Hoes 2004) and (Stuijk 2007) respectively) on a three-node MPSoC. The SDF graph for the H263 decoder has been presented earlier in this chapter. The SDF graph for the JPEG decoder is shown in Fig. 4.10. An FCFS scheduling policy was used in all the cases presented below. Table 4.2 shows the expected load on each processing node due to each application, if both applications achieve their required throughput.

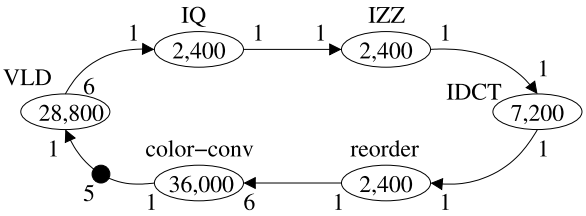


Fig. 4.10 SDF graph of JPEG decoder modeled from description in (Hoes 2004)

Table 4.2 Load (in proportion to total available cycles) on processing nodes due to each application

	H263	JPEG	Total
Processor 1	0.164	0.360	0.524
Processor 2	0.4	0.144	0.544
Processor 3	0.192	0.252	0.444
Total	0.756	0.756	1.512
Throughput Required	3.33e−6	5.00e−6	

The results were obtained after running the simulation for 100 million cycles. Figure 4.11 shows the performance of the two applications when they are run in isolation on the platform and also when they are run concurrently. In this figure, the resource manager does not interfere at all, and the applications compete with each other for resources. As can be seen, while the performance of H263 drops only marginally (depicted by the small arrow in the graph), a huge drop is observed in JPEG performance (big arrow in the graph). In fact, we see that even though the total load on each processing node is close to 50%, JPEG throughput is much lower than desired.

Figure 4.12 shows how a resource manager interferes and ensures that both applications are able to meet their minimum specified throughput. In this figure, the resource manager checks every 5 million cycles whether applications are performing as desired. Every time it finds that either JPEG or H263 is performing below the desired throughput, it suspends the other application. Once the desired throughput is reached, the suspended application is re-activated. We observe that the RM effectively interleaves three *infeasible* schedules (*JPEG alone*, *H263 alone*, and *H263/JPEG together* in Fig. 4.11) that yields a *feasible* overall throughput for each application. (In *alone*, only one application is active and therefore, those schedules are infeasible for the other application.)

Figure 4.13 shows the performance of applications when the sample period of resource manager is reduced to 500,000 cycles. We observe that progress of applications is *smoother* as compared to Fig. 4.12. The *transient phase* of the system is also shorter, and the applications soon settle into a *long-term average throughput*, and do not vary significantly from this average. This can be concluded from the almost horizontal curve of the achieved throughput. It should be mentioned that this

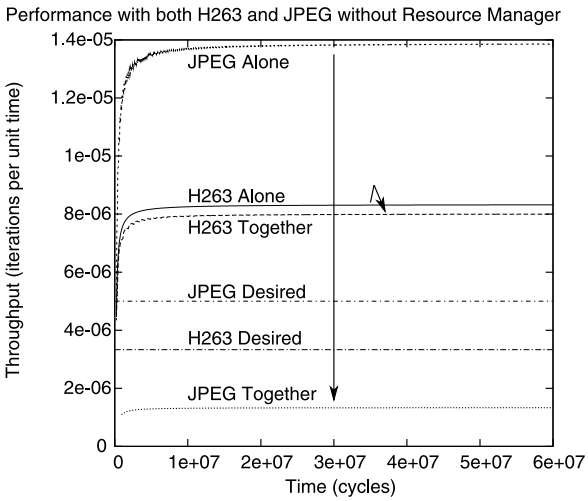


Fig. 4.11 Progress of H263 and JPEG when they run on the same platform – in isolation and concurrently

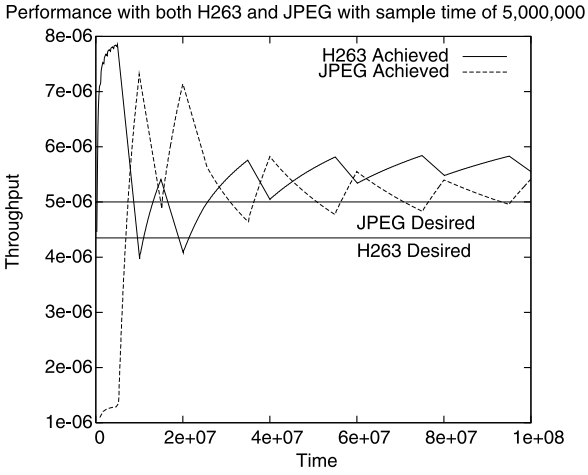


Fig. 4.12 With a resource manager, the progress of applications is closer to desired performance

benefit comes at the cost of increased monitoring from the resource manager, and extra overhead in reconfiguration (suspension and re-activation).

Table 4.3 shows the number of iterations that would be completed for each application in the simulation time, if both applications would achieve the desired throughput. The table also shows the number of iterations that are measured with and without intervention from the RM. The third column clearly indicates that JPEG executes only about one-fourth of the required number of iterations, whereas H263 executes about twice as often as needed. The last three columns demonstrate the use of the RM to satisfy the required throughput for both the applications. The last row indicates that the utilization of resources increases with finer grain of control from the RM.

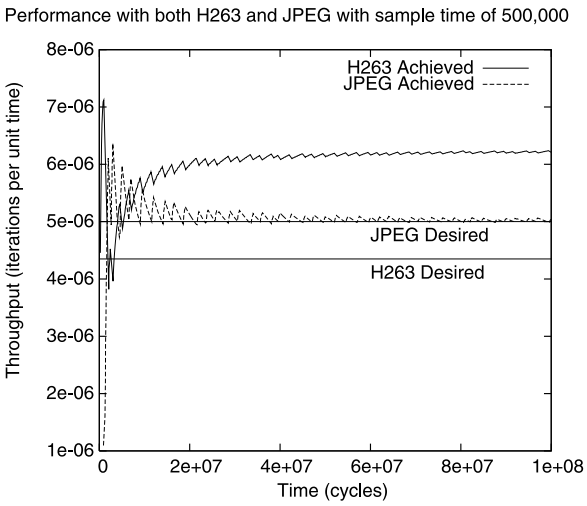


Fig. 4.13 Increasing granularity of control makes the progress of applications smoother

Table 4.3 Iteration count of applications and utilization of processors for different sampling periods for 100M cycles

	Specified	Without RM	RM sampling period		
			5,000k	2,500k	500k
H263 [# Iterations]	333	800	554	574	620
JPEG [# Iterations]	500	133	541	520	504
Processor 1 Util.	0.524	1.00	0.83	0.85	0.90
Processor 2 Util.	0.544	0.56	0.71	0.71	0.72
Processor 3 Util.	0.444	0.46	0.55	0.55	0.56
Total Utilization	1.512	2.02	2.09	2.11	2.18

Predictability Through Suspension

In this section we shall look at some results of statically computing the ratio of time to be spent in different states in order to achieve the desired throughput for all the applications. We take the same two application models – JPEG and H263 decoders as in the earlier experiments. Table 4.4 shows the time weights needed to achieve the desired performance. The minimum throughput needed for both applications is fixed at 500 iterations in 100 million cycles. The estimates of performance of the two applications in individual states – C0 to C3 is the same as obtained earlier using simulation. The linear programming analysis gives the time distribution as shown in Table 4.4. As can be seen in the table, the total time fraction is 0.92. Thus, only 92% of the total time is sufficient to achieve the desired throughput. For the remaining time, a number of options are available: (1) the empty state C0 can be used i.e. no application is active for the remaining time, (2) the time weights of the active states

Table 4.4 Time weights statically computed using linear programming to achieve desired performance

	C0	C1	C2	C3	Total
JPEG	0	1388	0	134	
H263	0	0	833	801	
Time Weight	0	0.3	0	0.62	0.92
JPEG Effective	0	416.35	0	83.65	**500**
H263 Effective	0	0	0	500	**500**

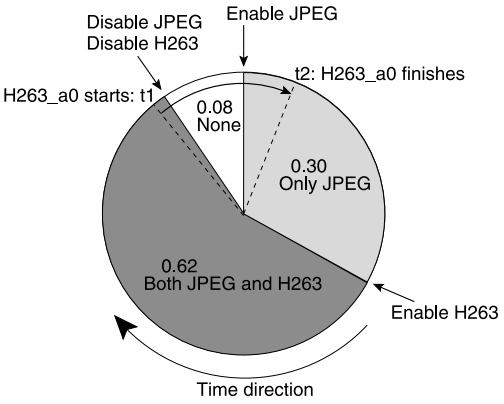


Fig. 4.14 The time wheel showing the ratio of time spent in different states

can be scaled such that the total is 1, and (3) the time spent in C3 can be increased such that both applications are active in the remaining time. Clearly the performance in second and third option is likely to be higher than the first option. However, when power is of concern then the first option is likely to be the best. In this section, we will see some results of all the options.

Another design parameter is the duration of the time wheel. This duration determines the precise time of switching from one combination to another. A smaller time wheel implies more switching, but with the potential gain of more uniform application behaviour. However, a bigger time wheel implies that the performance predicted by the linear programming solution is more accurate. However, with a smaller time wheel, the exact time spent in different combinations may vary from the desired time significantly. This comes from the fact that we have a non-preemptive system. This is explained using Fig. 4.14. The figure shows a time-wheel showing the ratio of time that is desired to be spent in different combinations. From Table 4.4 we know that only JPEG should be enabled for 0.30 fraction of time, and both JPEG and H263 for 0.62. This is realized using disable and enable commands sent from the resource manager to the processors. However, a disable command indicates that no more actors of that application may be executed. It does not immediately interrupt the ongoing execution of the actor. In Fig. 4.14, for example, if the actor $H263\_a0$ starts executing just before the application H263 is disabled, it may still continue

well into the phase when H263 is disabled. t_1 denotes the time that this actor starts and t_2 when it stops. As can be seen, most of the execution of $H263\_a0$ is in the phase when H263 is disabled. This creates a problem in achieving the performance exactly as predicted. In fact, if there is not sufficient time between disabling and enabling the application, the suspension may not have an effect at all.

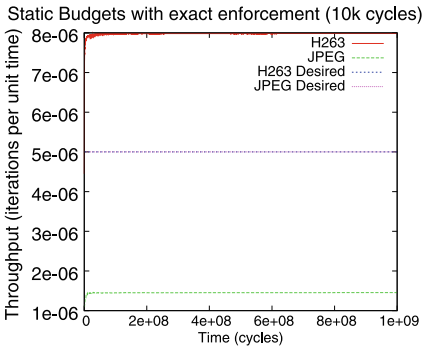
Figure 4.15(a) shows the performance of applications with time when the extra time is used for the combination C0 i.e. no application executing, when the time wheel is set to 10 thousand time units. We see that the performance is the same as without any suspension at all (see Fig. 4.11). Figure 4.15 shows that as the time wheel is increased, the performance of both applications improve and in most cases are above the desired level. The only anomaly is with the time wheel of 10 million time units (Fig. 4.15(e)) when the performance of H263 is just a little bit below the desired level. This again comes from the fact that when application H263 is enabled, it may not immediately start executing since actors of JPEG may still be queued on the processors. Figure 4.16 shows the performance when the time wheel is set to 10 million time units, for the cases when the extra time is used for the combination in which both applications are active, and for the case when the time weights are uniformly scaled up so as to not leave any spare time. As can be seen, the performance of both the applications is now above the desired level in both the cases.

5 Suggested Readings

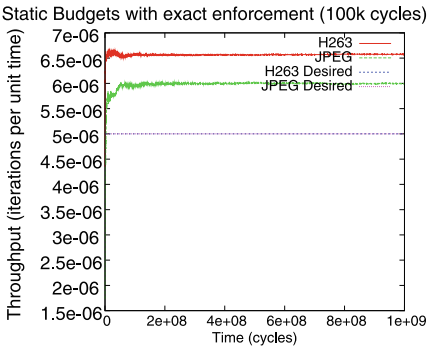
For traditional systems, with a single general-purpose processor supporting preemption, the analysis of schedulability of task deadlines is well known (Liu and Layland 1973) and widely used. Non-preemptive scheduling has received considerably less attention. It was shown by Jeffay et al. (Jeffay et al. 1991) and further strengthened by Cai and Kong (Cai and Kong 1996) that the problem of determining whether a given periodic task system is non-preemptively feasible even upon a single processor is already intractable. Also, research on multiprocessor real-time scheduling has mainly focused on preemptive systems (Davari and Dhall 1986; Baruah et al. 1996; Masrur et al. 2010).

Recently, more work has been done on non-preemptive scheduling for multiprocessor systems (Baruah 2006). Alternative methods have been proposed for analyzing task performance and resource sharing. A formal approach to verification of MPSoC performance has been proposed in (Richter et al. 2003). Use of real-time calculus for schedulability analysis was proposed in (Thiele et al. 2000). Computing worst-case waiting time, taking resource contention into account, for round-robin and TDMA scheduling (requires preemption) has also been analyzed (Hoes 2004). However, potential disadvantages of these approaches are that the analysis can be very pessimistic as has been explained in Chap. 2.

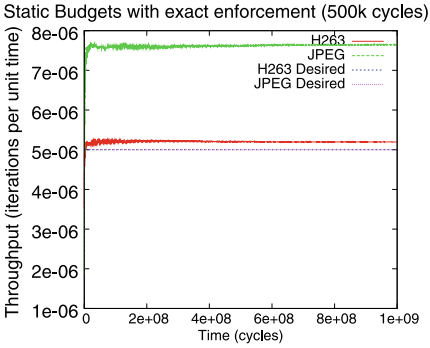
A lot of work has been done in the context of resource management for multiprocessor systems (Moreira et al. 2007; Nollet et al. 2008; Kumar et al. 2006a). The work in (Moreira et al. 2007) only considers preemptive systems, while our work is



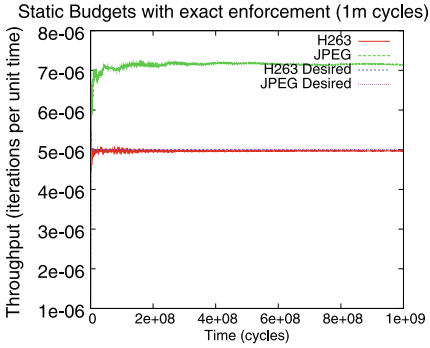
(a) Time wheel of 10 thousand time units



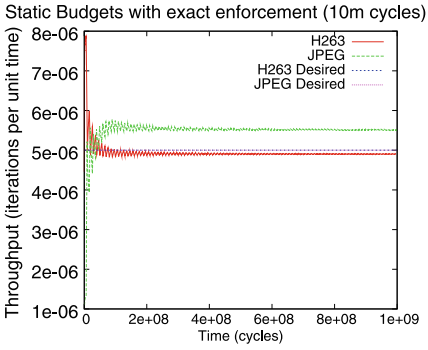
(b) Time wheel of 100 thousand time units



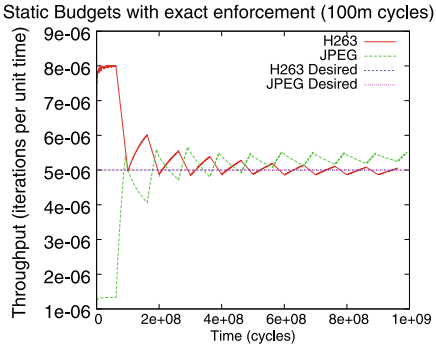
(c) Time wheel of 500 thousand time units



(d) Time wheel of 1 million time units



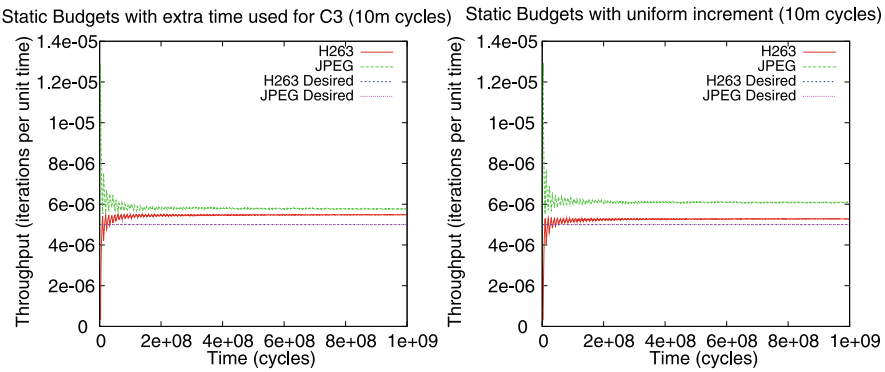
(e) Time wheel of 10 million time units



(f) Time wheel of 100 million time units

Fig. 4.15 Performance of applications H263 and JPEG with static weights for different time wheels. Both applications are disabled in the spare time, i.e. combination C0 is being used

targeted at non-preemptive systems. Non-preemptive systems are harder to analyze since the interference of other applications has to be taken into account. The work in (Nollet et al. 2008) presents a run-time manager for MPSoC platforms, but they



(a) Extra time for C3 (b) Weights uniformly incremented

Fig. 4.16 Performance of applications H263 and JPEG with time wheel of 10 million time units with the other two approaches

only consider one task mapped on one processor in the system; they do not allow sharing of processors. In (Kumar et al. 2006a) the authors deal with non-preemptive heterogeneous platforms where processors are shared, but only discuss the issue of budget enforcement and not of admission control.

The authors in (Paul et al. 2006) motivate the use of a scenario-oriented (or *use-case* in this book) design flow for heterogeneous MPSoC platforms. They propose to analyze the scenarios at design-time. However, with the need to add applications at run-time, a design-flow is needed that can accommodate this dynamic addition of applications. We presented such a flow in this chapter.

A number of techniques are present in literature to do the partitioning of program code into tasks. Compaan is one such example that converts sequential description of an application into concurrent tasks by doing static code analysis and transformation (Stefanov et al. 2004). Sprint also allows code partitioning by allowing the users to tag the functions that need to be split across different actors (Cockx et al. 2007). Yet another technique has been presented that is based on execution profile (Rul et al. 2007).

Table 4.5 shows a comparison of various analysis techniques that have been presented so far in literature, and where our approach is different. As can be seen, all of the research done in multiprocessor domain provides low utilization guarantees. Our approach on the other hand aims at achieving high utilization by sacrificing hard guarantees.

6 Conclusions

In this chapter, we presented a novel flow for designing systems with multiple applications, such that the entire analysis remains composable. This allows easy and quick analysis of an application-mix while properties of individual applications are

Table 4.5 Summary of related work (heterogeneous property is not applicable for uniprocessor schedulers)

Properties	Liu 1973	Jeffay 1991	Baruah 2006	Richter 2003	Hoes 2004	Our method
Multiprocessor	No	No	Yes	Yes	Yes	Yes
Heterogeneous	N.A.	N.A.	No	Yes	Yes	Yes
Non-preemptive	No	Yes	Yes	Yes	Yes	Yes
Non-periodic support	No	Yes	No	Yes	Yes	Yes
Utilization	High	High	Low	Low	Low	High
Guarantee	Yes	Yes	Yes	Yes	Yes	No

derived in isolation. Our flow also allows addition of applications at run-time, even when they are not known at design-time. We present a hybrid approach where the time-consuming application-specific computations are done at design-time, and in isolation to other applications – to maintain composability, and the use-case-specific computations are performed at run-time. Further, we present a resource manager (RM) for run-time aspects of resource management.

We propose a resource manager (RM) for non-preemptive heterogeneous MP-SoCs. Although the scheduling of these systems has been considered in the literature, the actual resource management in the context of concurrently executing applications is still unexplored area. Theoretically, design-time analysis of all possible use-cases can provide performance guarantees, but the potentially large number of use-cases in a real system and dynamic properties of applications make such analysis infeasible. Our resource manager shifts the burden of design-time analysis to run-time monitoring and intervention when necessary.

A high-level simulation model of the resource manger has been developed using POOSL methodology. A case study with an H263 and a JPEG decoder demonstrates that RM intervention is essential to ensure that both applications are able to meet their throughput requirements. Further, a finer grain of control increases the utilization of processor resources, and leads to a more stable system. Results of statically computing the time weights for suspension are also presented, although more research needs to be done to evaluate the size of time-wheel.

Chapter 5
Multiprocessor System Design and Synthesis

In the earlier chapters, we saw how to analyze performance of multiple applications executing concurrently on a multiprocessor platform. We saw how we can manage the resources available on a multiprocessor platform by splitting the management into design-time and run-time. We also saw how to use the performance analysis technique for run-time admission control of applications, and the budget enforcement technique to ensure all applications can achieve their desired performance. While analysis and management of a multiprocessor system are important, designing and programming the system are not easy either. With reducing time-to-market, designers are faced with the challenges of designing and testing systems quickly. Rapid prototyping has become very important to evaluate design alternatives efficiently, and to explore hardware and software alternatives quickly. Unfortunately, the lack of automated techniques and tools implies that most work is done by hand, making the design-process error-prone and time-consuming. This also limits the number of design-points that can be explored. While some efforts have been made to automate the flow and raise the abstraction level, these are still limited to single-application designs.

Figure 5.1 shows an ideal design flow to overcome this challenge. It is every designer's dream to directly input the application specification in the form of C-code (or whichever language it is given) – be it sequential or parallel, and generate and synthesize the best multiprocessor system that meets all the constraints of application performance, and at the same time achieve the design-objectives of low silicon area and power. Current state-of-the-art tools only allow for single-application designs, as shown by the shaded area in Fig. 5.1.

In this chapter, we present *MAMPS Multi-Application Multi-Processor Synthesis* – a design-flow that takes in application(s) specifications and generates the entire MPSoC, specific to the input application(s) together with corresponding software projects. This automated synthesis is indicated by the dashed box in Fig. 5.1. This allows the design to be directly implemented on the target architecture. For this flow, we assume that the applications are already partitioned and analyzed, and their SDF models are available. In this chapter we limit ourselves to a single use-case. The next chapter explains how multiple use-cases can be merged and partitioned for a system design.

A. Kumar et al., *Multimedia Multiprocessor Systems*, Embedded Systems,
DOI 10.1007/978-94-007-0083-3_5, © Springer Science+Business Media B.V. 2011

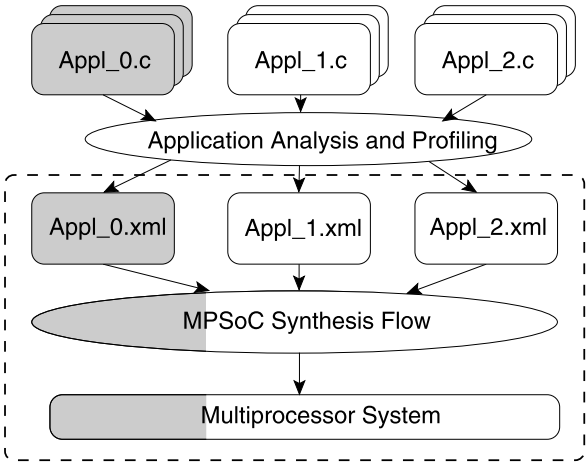


Fig. 5.1 Ideal design flow for multiprocessor systems

The flow presented here is unique in two aspects: (1) it allows fast DSE by automating the design generation and exploration, and (2) it supports multiple applications. To the best of our knowledge, there is no other existing flow to automatically map multiple applications to an MPSoC platform. The design space increases exponentially with increasing number of applications running concurrently; our flow provides a quick solution to that. To ensure multiple applications are able to execute concurrently, (1) we use non-blocking reads from and writes to buffers that do not cause deadlock even with multiple applications, (2) we have an arbiter that skips actors that are non-ready, and (3) we map channels to individual FIFOs to avoid head-of-line blocking.

The flow is used to develop a tool to generate designs targeting Xilinx platform FPGAs. FPGAs were selected as the target architecture as they allow rapid prototyping and testing. This *MAMPS* tool is made available online for use by the research community at (MAMPS 2009). In addition to a website, an easy to use GUI tool is also available for both Windows and Linux. The tool is used to generate several multiple-application designs that have been tested on Xilinx University Virtex II Pro Board (XUPV2P) (Xilinx 2010). However, the results obtained are equally valid on other FPGA architectures, and the tool can be easily extended to support other FPGA boards and architectures. We present a case study on how our methodology can be used for design space explorations using JPEG and H263 decoders. We were able to explore 24 design points that trade-off memory requirements and performance achieved with both applications running concurrently on an FPGA in a total of 45 minutes, including synthesis time.

This chapter is organized as follows. Section 1 provides an overview of the methods used for performance evaluation. Section 2 gives an overview of our flow, *MAMPS*, while Sect. 3 describes the tool implementation. Section 4 presents results of experiments done to evaluate our methodology. Section 5 provides some suggestions for further readings in the area of architecture-generation and synthesis flows

for multiprocessor systems. Section 6 concludes this chapter and gives directions for future work.

1 Performance Evaluation Framework

Performance evaluation forms an integral part of system design. In MPSoC designs, the design space exploration and the parameter optimization can quickly become intractable (Jerraya and Bacivarov 2006). Performance evaluation approaches can broadly be divided in two main categories – *statistical* and *deterministic*. Statistical approaches rely on developing a high-level system-performance model, calibrating the model parameters, and using them to predict the behaviour of new applications. The exact application is not taken into account. Deterministic approaches on the other hand, take the application into account. System simulation is one of the most common ways for deterministic performance evaluation. It relies on the execution of the complete system using input use-cases. Accuracy of this approach depends on the parameters covered in the simulation model. Unfortunately, the accuracy is generally inversely proportional to the speed. The greater the number of parameters modeled, the slower it is.

Analytical approach, also deterministic, is often used to investigate system capabilities. The sub-system is abstracted away using algebraic equations. Mathematical theories allow full analysis of the system performance at an early design stage. The probabilistic analysis provided in Chap. 3 is an example of such an approach. Another deterministic approach is measurement on the actual platform, but this is generally harder since the actual system is available much later in the design. This is exactly where our approach helps.

Our method allows mapping of applications on real hardware and measuring their performance accurately. This can be done for either individual or multiple applications, as desired. Application(s)-specific architecture is generated, and real performance can be measured way before the actual system is available without losing much accuracy. The design runs on an FPGA platform and the interaction between multiple applications can be observed.

Table 5.1 shows a comparison of using different techniques for deterministic evaluation. Analytical approaches are fast but generally lack accuracy. Simulation can improve on the accuracy part with enough parameters being modeled, but takes a long time before giving meaningful results. In order to speed up the simulation, some compute-intensive parts of simulation are often off-loaded to real hardware – a term known as hardware-acceleration. This approach is traditionally rather difficult to integrate in the simulation platform. However, it does provide high accuracy and is fast.

Our approach generates an MPSoC design for FPGA where actual performance can be measured. It is an example of *hardware emulation*, defined as the process of imitating the behaviour of one or more pieces of hardware, with another piece of hardware. While generally the goal is debugging, in our case it is performance

Table 5.1 Comparison of various methods to achieve performance estimates

	Time	Accuracy	Ease of use
Simulation	−	+	+
Analysis	+	−	+
H/w acceleration	+	+	−
Hardware Emulation	+	+	+

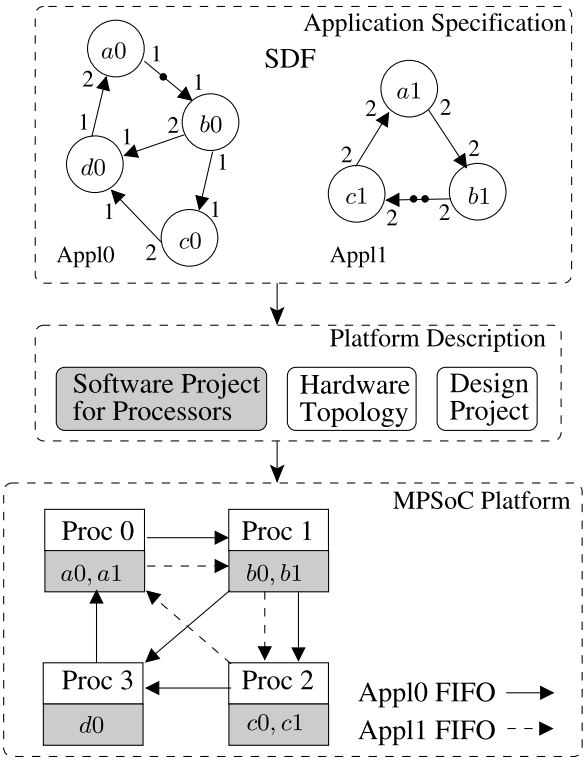


Fig. 5.2 MAMPS design flow

evaluation. FPGA multiprocessor design can quickly provide an estimate of performance of multiple applications which are sharing resources. While most of the other approaches focus only on a single application, we develop designs for multiple applications.

2 *MAMPS* Flow Overview

In this section, we present an overview of Multi-Application Multi-Processor Synthesis (*MAMPS*). Figure 5.2 shows an overview of our design flow. The application-descriptions are specified in the form of SDF graphs, which are used to generate

```
<application id="H263">
  <actor name="VLD">
    <port name="MotionComp" type="in" rate="1"/>
    <port name="IQ" type="out" rate="594"/>
  </actor>
  <actor name="IQ">
    <port name="VLD" type="in" rate="1"/>
    <port name="IDCT" type="out" rate="1"/>
  </actor>
  ...
  <channel name="VLD_IQ" srcActor="VLD" srcPort="IQ" dstActor="IQ"
                         dstPort="VLD" initialTokens="0"/>
  ...
```

Fig. 5.3 Snippet of H263 application specification

the hardware topology. The software project for each core is produced to model the application(s) behaviour. The project files specific to the target architecture are also produced to link the software and hardware topology. The desired MPSoC platform is then generated.

For example, in Fig. 5.2, two example applications *Appl*0 and *Appl*1 are shown with 4 and 3 actors respectively. From these graphs, *MAMPS* generates the desired software and hardware components. The generated design in this example, has four processors with actors $a0$ and $a1$ sharing *Proc*0, while $d0$ being the only actor executing on *Proc*3. The corresponding edges in the graphs are mapped to FIFO (first-in-first-out) channels as shown.

The flow can be easily used to design multiprocessor systems that support multiple applications. The target platform can be either FPGA or even an ASIC design. The current tool implemented uses Xilinx tool-chain (explained more in Sect. 3). The target architecture in this tool is Xilinx Virtex II Pro FPGAs. Even for designs that target ASIC platforms, our tool is useful for doing rapid prototyping and performance evaluation, since it can take a very long time before the final ASIC chip is available.

Application Specification

Application specification forms an important part of the flow. The SDF graphs of applications are specified in *xml* format. A snippet of the application specification file for H263 decoder is shown in Fig. 5.3. The corresponding SDF graph is shown in Fig. 5.4. The application here has been modeled from the data presented in (Hoes 2004). The figures illustrate how easy it is to write the application specification.

While the specification above is obtained through application profiling, it is also possible to use tools to obtain the SDF description for an application from its code directly. Compaan (Stefanov et al. 2004) is one such example that converts sequential description of an application into concurrent tasks. (It actually converts a sequential application into a limited set of KPN graph, namely cyclo-static dataflow graphs (CDFG).) These can then be converted into SDF graphs easily.

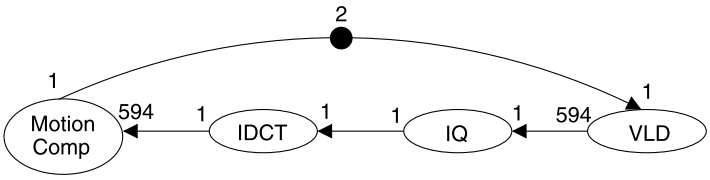


Fig. 5.4 SDF graph for H263 decoder application

The specification file contains details about how many actors are present in the application, and how they are connected to each other. The execution time of the actors and their memory usage on the processing core are also specified. For each channel present in the graph, the file describes whether there are any initial tokens present on it. The buffer capacity of a particular channel is specified as well.

When multiple applications are to be mapped to a common architecture, our flow allows use-case depiction. Very often in a given system, the system might support multiple applications, but only a few of them might be active at a given point in time. The use-case information may be supplied at design time together with application specification. This is explained in more detail in Chap. 6.

Functional Specification

When an application specification also includes high-level language code corresponding to actors in the application, this source code can be automatically added to the desired processor. An interface is defined such that SDF behaviour is maintained during execution. The number of input parameters of an actor function is equal to the number of incoming edges and the number of output parameters is equal to the number of output edges. The interface is shown in Fig. 5.5. $Token * in\_i$ indicates an array of input tokens consumed from i-th incoming edge, where the array length is equal to the consumption rate on that edge. Similarly, $Token * out\_i$ is an array of output tokens that are written during one execution of an actor. The application *xml* file indicates the function name that corresponds to the application actor. Figure 5.6 shows an example for the VLD actor of H263 application shown earlier. The function has an input channel from the MotionEstimation module and the data produced during execution is written to the output channel to InverseQuantization module. Therefore, the function definition of this actor only has one input and one output token pointer.

Platform Generation

From the *xml* descriptions, the platform description is generated. In case the architecture description and the mapping of actors to different processors is already

```
/*Functional definition of any function*/
void <functionName>(Token *in_1, Token *in_2, ..., Token *in_N,
                    Token *out_1, Token *out_2, ..., Token *out_M){
    ...
    ...
}
```

Fig. 5.5 The interface for specifying functional description of SDF-actors

```
<actorProperties actor="VLD">
    <processor type="p1">
    <executionTime time="120000"/>
    <functionName funcname="vld_c"/>
    <memory>
        <statesize max="100"/>
    </memory>
</actorProperties>

/*File vld_c.c: Functional description of vld_c */
void vld_c(Token *in_motion, Token *out_iq){
    ...
    ...
}
```

Fig. 5.6 Example of specifying functional behaviour in C

provided, mapping is done according to the specification. In other cases, the architecture is automatically inferred from the application specification(s). For a single application, each actor is mapped on a separate processor node, while for multiple applications, nodes are shared among actors of different applications. The total number of processors in the final architecture corresponds to the maximum number of actors in any application. For example, in Fig. 5.2, a total of 4 processors are used in the design. For processors that have multiple actors mapped onto them, an arbitration scheme is also generated.

All the edges in an application are mapped to a unique FIFO channel. This creates an architecture that mimics the applications directly. Unlike processor sharing for multiple applications, the FIFO links are dedicated as can be seen in Fig. 5.2. Since we have multiple applications running concurrently, there is often more than one link between some processors. Even in such cases, multiple FIFO channels are created. This avoids head-of-line blocking that can occur if one FIFO is shared for multiple channels (HOL 2009). Further, multiple channels reduce the sources of contention in the system.

As mentioned in Chap. 2, firing of an actor requires sufficient input tokens to be present on all its incoming edges. This implies that an actor might not be able to execute if any of the incoming buffers does not have sufficient tokens. The same holds when the output buffers of an actor are full. While this does not cause any problem when only one actor is mapped on a node, in the case of multiple actors, the other possibly *ready* actors might not be able to execute while the processor sits idle. To avoid this, non-blocking reads and writes are carried out, and if any read or write is unsuccessful, the processor is not blocked, but simply executes the other

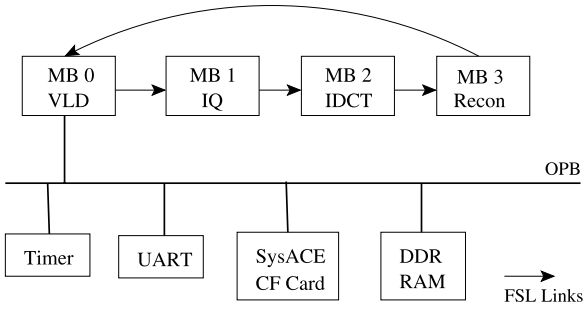


Fig. 5.7 Hardware topology of the generated design for H263

actor for which there are sufficient input tokens and buffer-space on all of its output edges.

3 Tool Implementation

In this section, we describe the tool we have developed. The actors in the MP-SoC flow are mapped to Microblaze processors (Xilinx 2010). The FIFO links are mapped on to Fast Simplex Links (FSLs). These are uni-directional point-to-point communication channels used to perform fast communication. The FSL depth is set according to the buffer-size specified in the application. PowerPC (Weiss and Smith 1994) processors are also supported in the flow. Customized communication blocks are generated to allow seamless communication between Microblaze and PowerPC as is published in (Shabbir et al. 2008, 2009).

An example architecture for H263 application is shown in Fig. 5.7. It consists of several Microblazes (MBs) with each actor mapped to a unique processor and additional peripherals such as Timer, UART, SysACE and DDR RAM. While the UART is useful for debugging the system, SysACE Compact Flash (CF) card allows for convenient performance evaluation by running continuously without external user interaction. The performance results are written to the CF card and they can be later retrieved using a card reader. Timer Module and DDR RAM are used for profiling the application and for external memory access respectively.

In this tool, in addition to the hardware topology, the corresponding software for each processing core is also generated automatically. For each processor, appropriate functions are inserted that model the behaviour of the tasks mapped on the processor. This can be a simple delay function if the behaviour is not specified. If the actual source code for the function is provided, it is added to the software description, as explained in Sect. 2. This also allows functional verification of applications on a real hardware platform. Routines for measuring performance, and sending results to the serial-port and to the on-board CF card are also generated.

Further, our software generation ensures that the tokens are read from (and written to) the appropriate FSL link in order to maintain progress, and to ensure correct

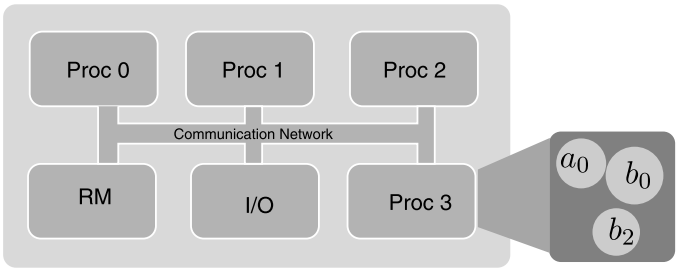


Fig. 5.8 Architecture with Resource Manager

functionality. In this way we avoid writing data to the wrong link which could easily throw the system in deadlock. Xilinx project files are automatically generated to provide the necessary interface between hardware and software components.

Resource Manager

The design is extended to allocate one processor for the resource manager (RM). Figure 5.8 shows the modified architecture when a resource manager is used in the system. The FIFO links in Fig. 5.7 are abstracted away with a communication fabric. The application description and properties computed off-line like the actor execution times, mapping and throughput expressions (as explained in Chap. 4) are stored in the CF card.

4 Experiments and Results

In this section, we present some of the results that were obtained by implementing several real and randomly-generated application SDF graphs using our design flow. Figure 5.9 shows the flow that is used to do the experiments. For each application, the buffer-sizes needed for the required performance are computed using SDF^3. These sizes are annotated in the graph description and used for the hardware flow described above. These buffer-sizes are modeled in the graph using a back-edge with the number of initial tokens on that edge equal to the buffer-size needed on the forward edge as is explained in Sect. 2 of Chap. 2. Further, we limit the auto-concurrency of actors to 1 since at any point in time, only one execution of an actor can be active. These constraints are modeled in the graph before the parametric throughput expressions are derived. Note that the graph used for computing the parametric expressions is not the same as the one that is mapped to architecture, but it leads to the same application behaviour since the constraints modeled in the graph come from the architecture itself.

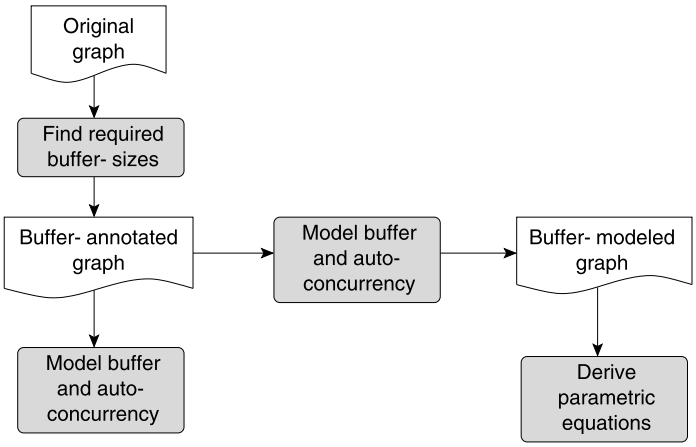


Fig. 5.9 An overview of the design flow to analyze the application graph and map it on the hardware

Reducing the Implementation Gap

The main objective of this experiment is to show that our flow reduces the implementation gap between system level and RTL level design. We show that our flow allows for more accurate performance evaluation using an emulation platform, compared to simulation (Theelen et al. 2007) and analysis. In addition, we present a case study using JPEG and H263 applications to show how our tool can be used for efficient design space exploration for multiple applications. The Xilinx XUP Virtex II Pro Development Board with an *xc2vp30* FPGA on-board was initially chosen as our implementation platform which was later upgraded to Xilinx Evaluation Kit ML605 with Virtex 6 LX240T FPGA that is shown in Fig. 5.10. As can be seen, a number of input/output options are available such that multiple devices can interact with the board, and interesting applications can be designed. All tools are now ported to Xilinx Design Suite 12.1 for synthesis and implementation. The newer versions of the corresponding tools can also use the generated designs by automatically upgrading them. All tools run on a Pentium Core at 3 GHz with 1 GB of RAM. Figure 5.11 shows the layout of the Virtex-6 LX240T FPGA for a design containing 100 Microblazes. Chip-area occupied by each Microblaze is highlighted for visibility. This was the biggest design we could synthesize before the FPGA ran out of Block RAMs. Logic on the other hand was only 20% utilized.

In order to verify our design flow, we generated 10 random application graphs using the SDF^3 tool (Stuijk et al. 2006a), and generated designs with 2 applications executing concurrently. Results of 10 such random combinations are summarized in Table 5.2. The results are compared with those obtained through simulation. We observe that in general, the application throughput measured on FPGA is about 8% lower than simulation. This is because the simulation model did not take into account the communication overhead. However, in some cases we observe that per-

Fig. 5.10 Xilinx Evaluation Kit ML605 with Virtex 6 LX240T (Xilinx 2010)

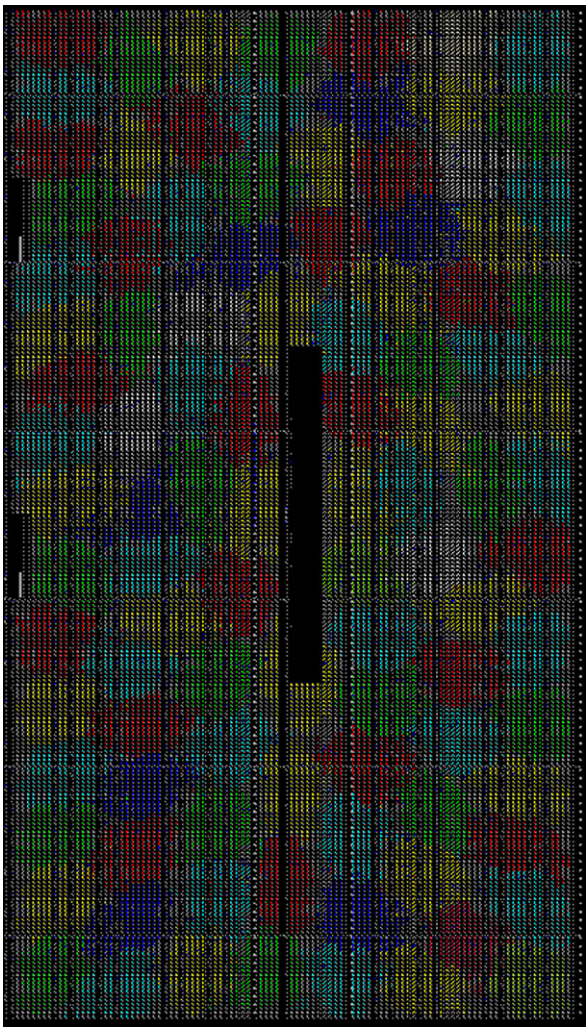


Fig. 5.11 (Colour online) Layout of the Virtex-6 FPGA with 100 Microblazes highlighted in colour

formance of some applications improved (shown in bold in Table 5.2). This is rather unexpected, but easily explained when going into a bit of detail.

Communication overhead leads to the actor execution taking somewhat longer than expected, thereby delaying the start of the successive actor. This causes the performance of that application to drop. However, since we are dealing with multiple applications, this late arrival of one actor might cause the other application to execute earlier than that in simulation. This is exactly what we see in the results. For the two use-cases in which this happens – namely A and C, the throughput of the other applications is significantly lower: 20 and 17 percent respectively. This also shows that the use-cases of concurrently executing multiple applications are more complex to analyze and reason about than a single application case.

Table 5.2 Comparison of throughput for different applications obtained on FPGA with simulation

Use-case	Appl 0			Appl 1		
	Sim	FPGA	Var %	Sim	FPGA	Var %
A	3.96	3.30	−20.05	1.99	2.15	**7.49**
B	3.59	3.31	−8.63	1.80	1.61	−11.90
C	2.64	2.74	**3.67**	1.88	1.60	−17.37
D	3.82	3.59	−6.32	0.85	0.77	−10.51
E	4.31	4.04	−6.82	1.44	1.35	−6.80
F	5.10	4.73	−7.75	0.51	0.48	−5.79
G	4.45	4.25	−4.55	1.11	0.97	−14.66
H	4.63	4.18	−10.65	1.16	1.05	−10.29
I	4.54	4.03	−12.48	2.27	2.13	−6.51
J	4.33	3.97	−8.92	1.08	1.00	−8.41
Average	–	–	−8.44	–	–	−8.29

DSE Case Study

Here we present a case study of doing a design space exploration and computing the optimal buffer requirement. Minimizing buffer-size is an important objective when designing embedded systems. We explore the trade-off between buffer-size used and the throughput obtained for multiple applications. For single applications, the analysis is easier and has been presented earlier (Stuijk et al. 2006b). For multiple applications, it is non-trivial to predict resource usage and performance, because multiple applications cause interference when they compete for resources. This is already shown in Table 5.2.

The case study is performed for JPEG and H263 decoder applications. The SDF models of the two applications are the same that were used in the previous chapter. In this case study, the buffer size has been modeled by the initial tokens present on the incoming edge of the first actor. The higher this initial-token count, the higher the buffer needed to store the output data. In the case of H263, each token corresponds to an entire decoded frame, while in the case of JPEG, it is the complete image.

Figure 5.12 shows how the throughput of JPEG decoder varies with increasing number of tokens in the graph. A couple of observations can be made from this figure. When the number of tokens (i.e. buffer-size in real application) is increased, the throughput also increases until a certain point, after which it saturates. When JPEG decoder is the only application running (obtained by setting the initial tokens in H263 to zero), we observe that its throughput increases almost linearly till 3. We further observe that increasing the initial tokens of H263 worsens the performance of JPEG, but only until a certain point.

The actual throughput measured for both applications is summarized in Table 5.3. Increasing initial tokens for H263 beyond 2 causes no change, while for JPEG the performance almost saturates at 4 initial tokens. This analysis allows the designer

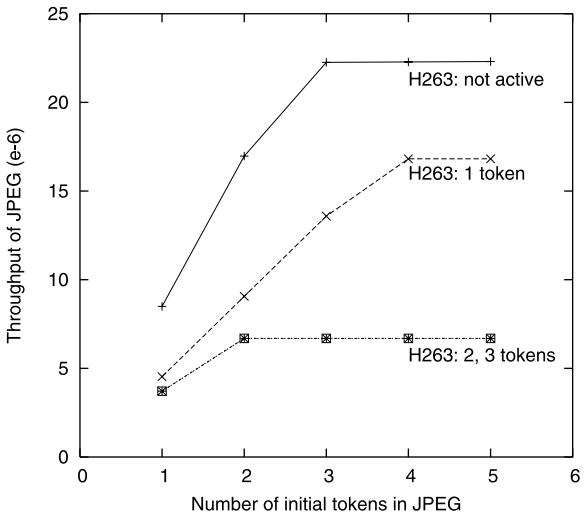


Fig. 5.12 Effect of varying initial tokens on JPEG throughput

Table 5.3 Number of iterations of the two applications obtained by varying initial number of tokens i.e. buffer-size, in 100 million cycles

H263	0		1		2		3	
JPEG	H263	JPEG	H263	JPEG	H263	JPEG	H263	JPEG
0	–	–	458	–	830	–	830	–
1	–	849	453	453	741	371	741	371
2	–	1697	453	906	669	669	669	669
3	–	2226	454	1358	669	669	669	669
4	–	2228	422	1682	669	669	669	669
5	–	2230	422	1682	669	669	669	669

to choose the desired performance-buffer trade-off for the combined execution of JPEG and H263.

Design Time

The time spent on the exploration is an important aspect when estimating the performance of big designs. The JPEG-H263 system was also designed by hand to estimate the time gained by using our tool. The hardware and software development took about 5 days in total to obtain an operational system starting from the application models using EDK. In contrast, our tool takes a mere 100 milli-seconds to generate the complete design. Table 5.4 shows the time spent on various parts of the flow. The Xilinx tools take about 36 minutes to generate the bit file together with the appropriate instruction and data memories for each core in the design.

Table 5.4 Time spent on DSE of JPEG-H263 combination

	Manual Design	Generating Single Design	Complete DSE
Hardware Generation	∼ 2 days	40 ms	40 ms
Software Generation	∼ 3 days	60 ms	60 ms
Hardware Synthesis	35:40 min	35:40 min	35:40 min
Software Synthesis	0:25 min	0:25 min	10:00 min
Total Time	∼ 5 days	36:05 min	45:40 min
Iterations	1	1	24
Time per iteration	∼ 5 days	36:05 min	1:54 min
Speedup	–	1	19

Our approach is very fast and is further optimized by modifying only the relevant software and keeping the same hardware design for different use-cases. Since the software synthesis step takes only about 25 seconds in our case study, the entire DSE for 24 design points was carried out in about 45 minutes. This hardware-software co-design approach results in a speed-up of about 19 when compared to generating a new hardware for each iteration. As the number of design points are increased, the cost of generating the hardware becomes negligible and each iteration takes only 25 seconds. The design occupies about 40% of logic resources on FPGA and close to 50% of available memory. This study is only an illustration of the usefulness of our approach for DSE for multiprocessor systems.

5 Suggested Readings

The problem of mapping an application to architecture has been widely studied in literature. One of the recent works that is most related to our research is ESPAM (Nikolov et al. 2006, 2008). This uses Kahn Process Networks (KPN) (Kahn 1974) for application specification. In our approach, we use SDF (Lee and Messerschmitt 1987) for application specification instead. Further, our approach supports mapping of multiple applications, while ESPAM is limited for single applications. Supporting multiple applications is imperative for developing modern embedded systems which support more than tens of applications on a single MPSoC. The same difference can be seen between our approach and the one proposed in (Jin et al. 2005) where an exploration framework to build efficient FPGA multiprocessors is proposed.

The Compaan/Laura design-flow presented in (Stefanov et al. 2004) also uses KPN specification for mapping applications to FPGAs. However, their approach is limited to a processor with a co-processor. Our approach aims at synthesizing complete MPSoC designs. Another approach for generating application-specific MPSoC architectures is presented in (Lyonnard et al. 2001). However, most of the steps in their approach are done manually. Exploring multiple design iterations is therefore

Table 5.5 Comparison of various approaches for providing performance estimates

	SDF^3	POOSL	ESPAM	*MAMPS*
Approach Used	Analysis	Simulation	FPGA	FPGA
Model Used	SDF	SDF	KPN	SDF
Single Appl	Yes	Yes	Yes	Yes
Multiple Appl	No	Yes	No	Yes
Speed	Fastest	Slow	Fast	Fast
Accuracy	Less	High	Highest	Highest
Hard RT Guarantee	Yes	No	No	No
Dedicated FIFO	N.A.	No	No	Yes
Arbiter Support	N.A.	Yes	N.A.	Yes
C-support	No	No	Yes	Yes

not feasible. In our flow, the entire flow is automated, including the generation of the final bit file that runs directly on an FPGA.

Yet another flow for generating MPSoC for FPGA has been presented in (Kumar et al. 2007). However, this flow focuses on generic MPSoC and not on application-specific architectures. Further, the work in (Kumar et al. 2007) uses networks-on-chip for communication fabric, while in our approach dedicated links are used for communication to remove resource contention. Another key difference is that processor sharing is not allowed across multiple applications.

Xilinx provides a tool-chain as well to generate designs with multiple processors and peripherals (Xilinx 2010). However, most of the features are limited to designs with only a bus-based processor-coprocessor pair with shared-memory. It is very time-consuming and error-prone to generate an MPSoC architecture and the corresponding software projects to run on the system. In our approach, MPSoC architecture is automatically generated together with the respective software projects for each core.

The MAMPS flow has been extended to support a communication assist (CA) in a multiprocessor systems (Shabbir et al. 2010). The use of a CA decouples the task of communication and computation. The processor can solely focus on computation while the communication is handled by the CA. The tool for automatically generating architectures with a CA is also included on the book website (MAMPS 2009).

Table 5.5 shows various design approaches that provide estimates of application performance. The first method uses SDF models and computes the throughput of the application by analyzing the application graph. However, it is only able to predict the performance of single applications. The simulation approach presented in (Kumar et al. 2006a) uses POOSL (Theelen et al. 2007) for providing application performance estimates. This is more accurate than high level formal analysis since more details can be modeled and their effects are measured using simulations. ESPAM is the closest to our approach as it also uses FPGA and supports functional description(s)

of application(s) in C. However, it does not support multiple applications. *MAMPS* supports multiple applications, and provides fast and accurate results.

6 Conclusions

In this chapter, a design-flow is presented to generate multiprocessor designs for multiple applications. The approach takes application(s) description(s) and produces the corresponding MPSoC system. This is the first flow that allows mapping of multiple applications on a single platform. The tool developed using this flow is made available online (MAMPS 2009). The flow allows the designers to traverse the design space quickly, thus making DSE of even concurrently executing applications feasible. A case study is presented to find the trade-offs between the buffer-size and performance when JPEG and H263 execute concurrently on a platform.

However, the number of applications that can be concurrently mapped on the FPGA is limited by the hardware resources present. When synthesizing designs with applications of 8–10 actors and 12–15 channels, we found that it was difficult to map more than four applications simultaneously due to resource constraints, namely block RAMs. A bigger FPGA would certainly allow bigger designs to be tested.

Chapter 6
Multiple Use-cases System Design

In the previous chapter, we have seen how to design and synthesize multiprocessor systems for multiple applications. As has been motivated in the earlier chapters, not all applications are always active at the same time. Each combination of simultaneously active applications is defined as a *use-case*. For example, a mobile phone in one instant may be used to talk on the phone while surfing the web and downloading some Java application in the background. In another instant it may be used to listen to mp3 music while browsing JPEG pictures stored in the phone, and at the same time allow a remote device to access the files in the phone over a bluetooth connection.

The number of such potential use-cases is exponential in the number of applications that are present in the system. The high demand of functionalities in such devices is leading to an increasing shift towards developing systems in software and programmable hardware in order to increase design flexibility. However, a single configuration of this programmable hardware may not be able to support this large number of use-cases with low cost and power. We envision that future complex embedded systems will be partitioned into several configurations and the appropriate configuration will be loaded into the reconfigurable platform on the fly as and when the use-cases are requested. This requires two major developments at the research front: (1) a systematic design methodology for allowing multiple use-cases to be merged on a single hardware configuration, and (2) a mechanism to keep the number of hardware configurations as small as possible. More hardware configurations imply a higher cost since the configurations have to be stored in the memory, and also lead to increased switching in the system.

In this chapter, we present a solution to the above-mentioned objectives. Following are the key contributions of this chapter:

- *Support for Multiple Use-cases:* An algorithm for merging use-cases onto a single (FPGA) hardware configuration such that multiple use-cases may be supported in a single configuration, while minimizing hardware resources.
- *Partitioning Use-cases:* When (FPGA) area constraints do not allow mapping of all use-cases on one configuration, a methodology to partition use-cases in a way that the number of partitions (or configurations of FPGA) is minimized.

A. Kumar et al., *Multimedia Multiprocessor Systems*, Embedded Systems,
DOI 10.1007/978-94-007-0083-3_6,

- *Reducing Complexity:* Use-case partitioning is an instance of the *set-covering problem* (Cormen et al. 2001), which is known to be *NP-hard*. We propose efficient heuristics to solve this problem and compare their performance and complexity.
- *Area Estimation:* A technique that accurately predicts the resource requirements on the target FPGA without going through the entire synthesis process, thereby saving DSE time.
- *MPSoC Design Tool for FPGA:* All of the above methods and algorithms are implemented, such that the entire multi-processor system can be generated for the given application and use-case descriptions in a *fully automated* way for Xilinx FPGAs. Besides the hardware, the required software for each processor is also generated. The tool is available at www.es.ele.tue.nl/mamps/ (MAMPS 2009).

The above contributions are essential to further research in design automation community since the embedded devices are increasingly becoming multi-featured. Our flow allows designers to generate MPSoC designs quickly for multiple use-cases and keep the number of hardware configurations to a minimum. Though the flow is aimed at minimizing the number of partitions, it also generates all the partitions and allows the designer to study the performance of all use-cases in an automated way. The designer can then tailor the partitions (e.g. change processor-arbiters) to achieve better performance of all applications in a use-case.

While the flow is suitable for both design and evaluation, in this book we focus on the suitability of our flow for evaluating whether all the applications can meet their functional requirements in all the use-cases on FPGA. We present a number of techniques to minimize the time spent in evaluation and design space exploration of the system. As before, we assume that applications are specified in the form of Synchronous Data Flow (SDF) graphs (Lee and Messerschmitt 1987; Sriram and Bhattacharyya 2000).

This chapter is organized as follows. Section 1 describes our approach of merging use-cases in a single hardware description, while Sect. 2 explains how our partitioning approach splits the use-cases when not all of them can fit in one design. Section 3 explains how resource utilization is estimated without synthesis. Section 4 presents results of experiments done to evaluate our methodology. Section 5 introduces related work for architecture-generation and synthesis flows for multiple use-cases. Section 6 concludes the chapter and gives directions for future work.

1 Merging Multiple Use-cases

In this section, we describe how multiple use-cases are merged into one design to save precious synthesis time and minimize hardware cost. When multiple use-cases are to be catered for during performance evaluation, time spent on hardware synthesis forms a bottle-neck and limits the number of designs that can be explored. When designing systems, the merging of multiple use-cases is even more important as it often reduces the hardware resources needed in the final platform. Further, the switching time between different use-cases is reduced substantially.

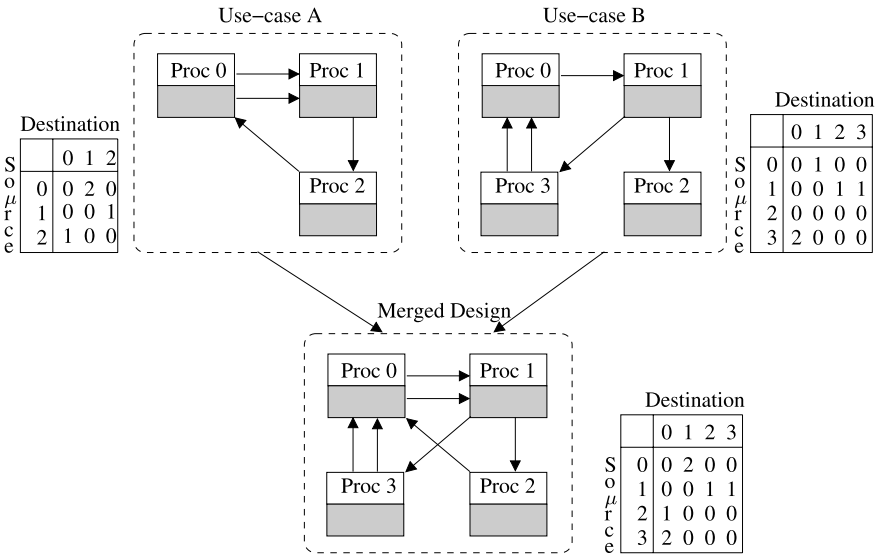


Fig. 6.1 An example showing how the combined hardware for different use-cases is computed. The corresponding communication matrix is also shown for each hardware design

Each application in the system requires hardware to be generated for simulation. Therefore, each use-case in turn has a certain hardware topology to be generated. In addition to that, software is generated for each hardware processor in the design that models the set of actors mapped on it. The following two sub-sections provide details of how the hardware and software are generated.

Generating Hardware for Multiple Use-cases

With different use-cases, since the hardware design is usually different, an entire new design has to be synthesized. Here we describe how we can merge the hardware required for different use-cases. Figure 6.1 shows two use-cases A and B, with different hardware requirements that are merged to generate the design with the *minimal* hardware requirements to support both. The combined hardware design is a super-set of all the required resources such that all use-cases can be supported. The key motivation for the idea comes from the fact that while multiple applications are active concurrently in a given use-case, different use-cases are active exclusively.

The complete algorithm to obtain the minimal hardware to support all the use-cases is described in Algorithm 2. The algorithm iterates over all use-cases to compute their individual resource requirements. This is in turn computed by using the estimates from the application requirements. While the number of processors needed is updated with a *max* operation – line 13 in Algorithm 2, the number of FIFO (first-in-first-out) buffers is added for each application – indicated by line 16. The total

Algorithm 2: GenerateCommunicationMatrix: determining minimal hardware design that supports multiple use-cases

Input: U_i // Description of which applications are in use-case U_i.
Input: A_i // SDF description of application A_i.
Output: N_{proc} // The total number of processors needed.
Output: X_{ij} // The total number of FIFO channels needed.

```
1:  // Let X_ij denote the number of FIFO channels needed from processor P_i
    to P_j
2:  X = 0 // Initialize the communication matrix to 0
3:  N_proc = 0 // Initialize the number of processors to 0
4:  for all Use-cases U_k do
5:      Y = 0 // Y_ij stores the number of FIFO channels needed for U_k
6:      N_proc,U_k = 0 // Initialize processor count for use-case U_k to 0
7:      for all Applications A_l do
8:          // Update processor count for U_k
9:          N_proc,U_k = max(N_proc,U_k, N_proc,A_l)
10:         for all Channels c in A_l do
11:             // Increment FIFO channel count
12:             Y_getProc(c_src)getProc(c_dest) = Y_getProc(c_src)getProc(c_dest) + 1
13:         end for
14:     end for
15:     N_proc = max(N_proc, N_proc,U_k) // Update overall processor count
16:     for all i and j do
17:         X_ij = max(X_ij, Y_ij)
18:     end for
19: end for
```

FIFO requirement of each application is computed by iterating over all the channels and adding a unique edge in the communication matrix for them. The communication matrix for the respective use-cases is also shown in Fig. 6.1.

To compute minimal hardware requirements for the overall hardware for all the use-cases, both the number of processors and the number of FIFO channels are updated by *max* operation (line 19 and 22 respectively in Algorithm 2). This is important because this generates only as many FIFO channels between any two processors as is maximally needed for any use-case. Thus, the generated hardware stays minimal. Therefore, in Fig. 6.1 while there are in total 3 FIFO channels between *Proc* 0 and *Proc* 1, at most two are used at the same time. Therefore, in the final design only 2 channels are produced between them.

Generating Software for Multiple Use-cases

Software compilation is a lot faster as compared to hardware synthesis in the *MAMPS* approach. The flow is similar to the one for generating software for sin-

gle use-cases. However, we need to ensure that the numbering for FIFO channels is correct. This is very important in order to ensure that the system does not go into deadlock. For example, in Fig. 6.1, *Proc* 0 in the merged hardware design has three incoming links. If we simply assign the link-ids by looking at the individual use-case, in *Use-case B* in the figure, the first link-id will be assigned to the channel from *Proc* 3. This will block the system since the actor on *Proc* 3 will keep waiting for data from a link which never receives anything.

To avoid the above situation, the communication matrix is first constructed, even when only the software needs to be generated. The link-ids are then computed by checking the number of links before the element in the communication matrix. For the output link-id, the numbers in the row are added, while for incoming links, the column is summed up. For example, in Fig. 6.1 the communication matrix of *Use-case B* suggests that the incoming links to *Proc* 0 are only from *Proc* 3, but the actual hardware design synthesized has one extra link from *Proc* 2. The incoming link-id should therefore take that into account in software.

Combining the Two Flows

Figure 6.2 shows how the hardware and software flows come together to get results for multiple use-cases quickly. The input to the whole flow is the description of all use-cases. From these descriptions, the communication matrix is constructed. This is used to generate the entire hardware. The same matrix is also used when software has to be generated for individual use-cases. The boxes that are shown in grey are repeated for each use-case. The flow terminates when all the use-cases are explored. The results of each use-case are fed from the FPGA board to the host computer via the serial port and are also written out on to the Compact Flash card. As can be seen, the hardware part is executed only once while the software part is iterated until results for all the use-cases are obtained. This flow makes execution of multiple use-cases a lot faster since hardware synthesis is no more a bottleneck in system design and exploration. It should be noted that in this flow, the FPGA is completely reconfigured after each use-case, even though the hardware used is identical. This overhead can be alleviated in one of the three ways: (1) by only reconfiguring the BRAMs for each software, (2) by having enough memory to accommodate software for all the use-cases, and (3) by having a loader in each processor that can load the required software for each use-case. Any of these above methods can be used to reduce the reconfiguration time. The ideas presented here are orthogonal to such techniques.

The use-case analysis (*Analyze All Use-cases*) is done to find the maximum number of use-cases that can fit in one hardware design. (This is done by our area estimation methods that are explained in Sect. 3.) Formally, given a set S of m use-cases $S = \{U_0, U_1, \ldots U_{m-1}\}$, we wish to determine the biggest possible sub-set of S that is *feasible*, where feasibility implies that all the elements of the set can be merged into one hardware design that can fit in the given FPGA device. The next section explains what happens when not all use-cases can be merged in one hardware design.

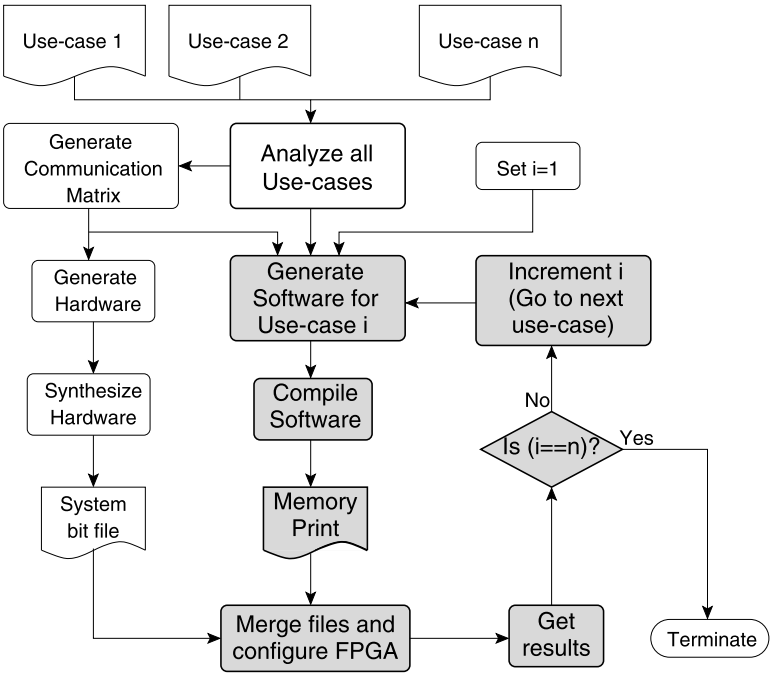


Fig. 6.2 The overall flow for analyzing multiple use-cases. Notice how the hardware flow executes only once while the software flow is repeated for all the use-cases

2 Use-case Partitioning

Resource is always a constraint, and FPGA devices do not escape from this rule. As the number of use-cases to be supported increases, the minimal hardware design increases as well, and it often becomes difficult to fit all use-cases in a single hardware design. Here we propose a methodology to divide the use-cases in such a way that all use-cases can be tested, assuming that all use-cases can at least fit in the hardware resources when they are mapped in isolation. (If an individual use-case does not fit, a bigger FPGA device is needed.) Further, we wish to keep the number of such hardware partitions as small as possible since each extra partition implies extra hardware synthesis time, and worse, switching time. This is an NP-hard problem as described below.

Problem 1 We are given $S = \{U_0, U_1, \ldots U_{m-1}\}$, where each use-case U_i is feasible in itself. Further, let us define set $\mathcal{F}$ of all feasible subsets of S. **Use-case partitioning** is finding the minimum subset $\mathcal{C} \subseteq \mathcal{F}$ whose members cover all of S.

Solution 1 This is clearly an instance of set-covering problem where the universe is depicted by S, and the subsets are denoted by $\mathcal{F}$. The set $\mathcal{C}$ we are looking for

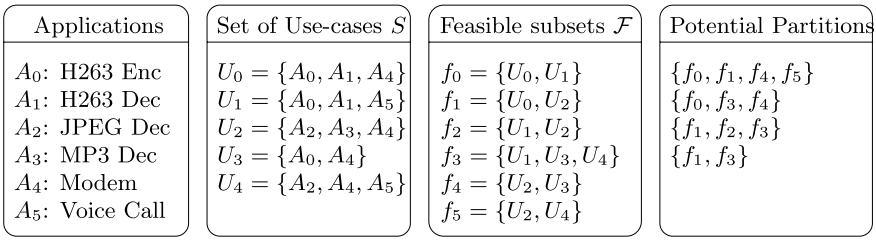


Fig. 6.3 Putting applications, use-cases and feasible partitions in perspective

is the solution of the minimum set-covering problem, and corresponds to that of the use-case partitioning problem. Each set in $\mathcal{C}$ corresponds to a feasible hardware partition. Set-covering problem is known to be NP-hard (Garey and Johnson 1979; Cormen et al. 2001). Any set-covering problem can be solved by translating it into the use-case partitioning problem. Use-case partitioning is therefore also an NP-hard problem.

The cost in both verification and design synthesis is directly proportional to the number of sets in $\mathcal{C}$. During verification, it is the time spent in synthesis which increases with partition count, while for system design more partitions imply a higher memory and switching cost. Since this is an NP-hard problem, in our tool we have used an approximation algorithm to solve it, called the greedy algorithm. The largest feasible subset of use-cases is first selected and a hardware partition is created for it. This is repeated with the remaining use-cases until all the use-cases are covered. As mentioned in (Cormen et al. 2001), the maximum approximation error in using this technique over the minimal cover is $\ln|X| + 1$, where X is the number of elements in the largest feasible set.

Figure 6.3 helps in understanding the partitioning problem better and provides a good perspective of the hierarchy of sets. The first box shows the applications that need to run on the platform. These are some of the applications that run on a mobile phone. The next box shows some of the use-cases that are typical, e.g. U_1 represents a video-call that requires video encoding, video decoding and regular voice call. As mentioned earlier, a use-case is a set of applications that run concurrently. The next box shows the family of sets $\mathcal{F}$, each of which is feasible. For simplicity only a part of $\mathcal{F}$ is shown in the figure. Clearly, the subsets of elements of $\mathcal{F}$ are also feasible e.g. when f_3 is feasible, so is $\{U_3, U_4\}$. Therefore, we should only include maximal subsets; $F - i$ is maximal if we cannot add any other use-case. As often the case, no feasible-set exists which contains all the use-cases. Therefore, a subset $\mathcal{C} \subseteq \mathcal{F}$ needs to be chosen such that all the use-cases are covered in this subset. A few of such possible subsets are shown in the last box. The last option is preferred over the rest since it provides only two partitions.

Hitting the Complexity Wall

In order to be able to implement the greedy algorithm, we still need to be able to determine the largest feasible set. This poses a big problem in terms of implementation. The total number of possible sets grows exponentially with the number of use-cases. Suppose, we have 8 applications in the system, and every combination of them is possible, we have 255 use-cases overall. Since each use-case can either be in the set or not, we obtain a total of 2^{255} sets. Each set then needs to be examined whether it is feasible or not – this takes linear time in size of the set, which can in the worst case be the number of applications in the system. Thus, a system with N applications and M possible use-cases, has a complexity of $O(N.2^M)$ to find the largest feasible set. In the worst-case, the number of use-cases is also exponential, i.e. $M = 2^N - 1$. We see how the design-space becomes infeasible to explore. In Sect. 4 we see some results of actual execution times.

Reducing the Execution Time

Here we present some measures to reduce the execution time. The following approaches do not reduce the complexity of the algorithm, but may provide significant reduction in execution time.

(1) *Identify infeasible use-cases:* Our intention is to be able to analyze all the use-cases that we can with our given hardware resources. Identifying the infeasible use-cases reduces the potential set of use-cases.
(2) *Reduce feasible use-cases:* This method identifies all the use-cases that are *proper* subsets of feasible use-cases – such use-cases are defined as *trivial*, while the ones that are not included in any other feasible use-case are defined as non-trivial. When a use-case is feasible, all its sub-sets are also feasible. Formally, if a use-case U_i is a subset of U_j, the minimal hardware needed for U_j is sufficient for U_i. (A proper subset here implies that all the applications executing concurrently in U_i are also executing in U_j, though the inverse is not true.) In other words, any partition that supports use-case U_j will also support U_i. It should be noted however that the performance of applications in these two use-cases may not be the same due to different set of applications active, and therefore it might be required to evaluate performance of both use-cases.

These approaches are very effective and may significantly reduce the number of feasible use-cases left for analysis. With a scenario of 10 randomly generated applications and 1023 use-cases (considering all the possibilities), we found that only 853 were feasible. (This depends on the available hardware resources. On our FPGA platform only 853 were feasible.) The reduction technique further reduced the number of non-trivial use-cases to 178. The above approaches reduce the execution time but do not help in dealing with complexity. However, the optimality of the solution (in generation of feasible sets, not in the set-cover) is maintained.

Algorithm 3: FirstFitSetUseCasePartitioning: partitioning use-cases using first fit algorithm – polynomial-complexity algorithm

Input: U_i // Description of which applications are in use-case U_i.
Output: *Partition*[ ] // Stores details of all the partitions.
Output: k // The number of partitions created.

```
1:  // Let tmp[][] and final[][] be communication matrices, initialized to zero
2:  // final[][] stores the matrix with all the use-cases that fit in the current
    partition
3:  // Initialize all use-cases as not done
4:  UseCaseDone[ ] = 0
5:  // Ignore the infeasible use-cases
6:  UseCaseDone[i] = −1 ∀i when U_i is infeasible
7:  // Reduction step
8:  UseCaseDone[i] = j + 2 ∀i when U_i is a sub-set of U_j
9:  // Partition[k] stores the use-cases that are assigned to the k-th partition
10: k = 0
11: while Use-cases left (Translates to UseCaseDone[i] = 0 for at least one i) do
12:     tmp_mn = 0 and final_mn = 0
13:     for all UseCases U_i with UseCaseDone[i] = 0 do
14:         tmp_mn = final_mn
15:         Update tmp_mn by merging UseCase U_i
16:         if tmp_mn fits in device then
17:             UseCaseDone[i] = 1
18:             Add i to Partition[k]
19:             final_mn = tmp_mn
20:         end if
21:     end for
22:     // Advance partition
23:     k = k + 1
24: end while
```

Reducing Complexity

In this section, we propose a simple heuristic to compute the partitions. This heuristic reduces the complexity significantly albeit at the cost of optimality. As mentioned earlier, the greedy approach of partitioning requires to compute the largest feasible set. Since computing the largest set has a high complexity, we have an alternative implementation which simply gives the first partition that includes the first non-included use-case, and scans the whole list to check which use-cases can be added such that the set remains feasible. The algorithm is shown in Algorithm 3. An array is maintained to check which use-cases are not yet included in any partition (*UseCaseDone*). Infeasible use-cases are indicated in line 6 in the algorithm. Use-cases are then reduced by considering only the non-trivial use-cases. Trivial

use-cases are assigned the identifier of its super-set (line 8). Note that in some cases there are multiple supersets. In such cases, the first one is chosen. This reduces the number of use-cases that need to be considered.

Partitions are then created on a first-come-first-serve basis. The order of use-cases in the input may therefore affect partitioning. As can be seen, once a use-case fits in a partition, it is not checked whether that is the optimal partition. In the worst-case, each use-case might result in its own partition, and the algorithm would then require $O(M)$ iterations of the *while* loop, each requiring M passes in the *for* loop. Therefore, the total complexity of this approach is $O(M^2)$ as compared to $O(2^M)$ in the original approach. Section 4 compares the execution times of the two approaches. Feasibility of a partition can be checked by conducting synthesis and checking whether the resources are sufficient. However, this is very time consuming. Therefore, in order to identify infeasible partitions (lines 6 and 16 in Algorithm 3), we use a quick area estimation technique that is explained in the next section.

3 Estimating Area: Does It Fit?

Whenever one talks about FPGA design, resource limitation is a major issue, and it is always important to know whether a desired design fits in limited FPGA resource. Especially because hardware synthesis takes so much time, if the design finally does not fit on a target architecture, a lot of precious time is wasted which makes an exploration considerably slower. In this section, we therefore provide the necessary area estimation formulae. Our experiments were done on Xilinx University Board containing a Virtex II Pro XC2VP30, and the same methodology can be applied to compute similar formulae for other target architectures as well. ISE 8.2i and EDK 8.2i were used for synthesis.

An FSL can be implemented either using block RAMs or using LUTs (lookup tables) in the FPGA. Each LUT in the Virtex II Pro series has 4 inputs. In the LUT implementation, the FIFO is synthesized using logic, while in BRAM implementation embedded dual-port block RAMs are used to synthesize these channels. Since both are crucial resources, we did the whole experiment with both these options. Following four sets of experiments were done.

- *Vary FSL, with BRAM:* Base design of one Microblaze and one FSL, incrementing FSL count to eight, with FSLs implemented using BRAMs.
- *Vary FSL, with logic:* Base design of one Microblaze and one FSL, incrementing FSL count to eight, with FSLs implemented using logic.
- *Vary Microblaze, with BRAM FSL:* Base design of one Microblaze and 8 FSLs in total, incrementing Microblaze count to eight, with FSLs implemented using BRAMs.
- *Vary Microblaze, with logic FSL:* Base design of one Microblaze and 8 FSLs in total, incrementing Microblaze count to eight, with FSLs implemented using logic.

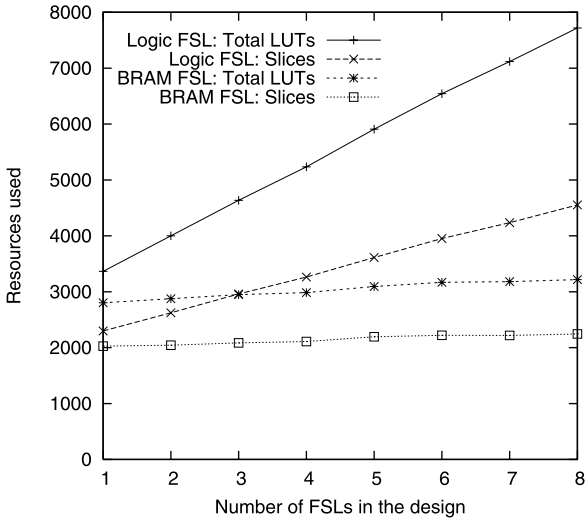


Fig. 6.4 Increase in the number of LUTs and FPGA Slices used as the number of FSLs in design is increased

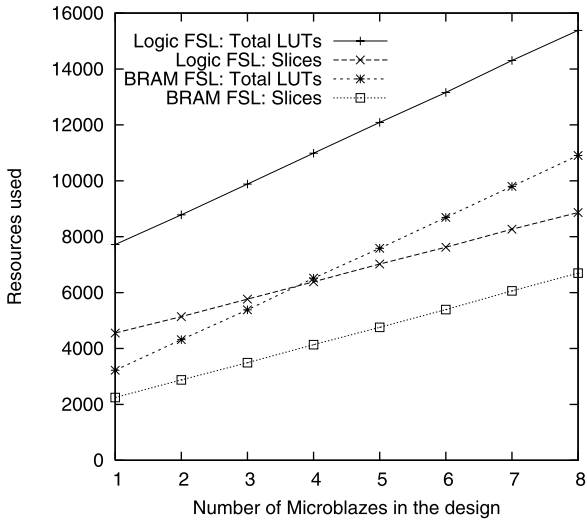


Fig. 6.5 Increase in the number of LUTs and FPGA Slices used as the number of Microblaze processors is increased

Each FSL was set to a depth of 128 elements. (It is also possible to set the FIFO depth in the specification, but we used a constant number for this study to minimize the number of variables.) For a 32-bit element this translates to 512-byte memory. A BRAM in this device can hold 2 kB of data, translating to 512 elements per BRAM. The number of slices, LUTs and BRAMs utilized was measured for all experiments. Results of the first two sets are shown in Fig. 6.4 and of the next two are shown in Fig. 6.5. The increase in the total logic utilized is fairly linear as expected. In Virtex II Pro family, each FPGA slice contains 2 LUTs, but often not

Table 6.1 Resource utilization for different components in the design

	Total	Base Design	Each Fast Simplex Link		Each Microblaze
			BRAM Impl	Logic Impl	
BRAM	136	0	1	0	4 (32)
LUTs	27392	1646	60	622	1099
Slices	13696	1360	32	322	636

both are used. Thus, we need to take slice utilization into account. LUT utilization is shown as the measure of logic utilized in the design.

Table 6.1 shows the resource utilization for different components in the design obtained by applying linear regression on the results of experiments. The second column shows the total resources present in XC2VP30. The next column shows the utilization for a basic design containing OPB (on-chip peripheral bus), the CF card controller, timer and serial I/O. The next two columns show the resources used for each dedicated point-to-point channel in the design.

The last column in Table 6.1 shows the same for Microblaze in the design. In our design one Microblaze is assigned the task of communicating with the host and writing the results to the CF card. This core was designed with a much bigger memory for instruction and data. It uses 32 BRAMs in total, translating to 32 kB memory each for data and instructions. The other cores have a much smaller memory at only 4 kB each for data and instructions.

It is easy to obtain the total resource count that will be utilized upon synthesis. In our tool we also output the same and use it as a means to estimate whether the design would fit in the given resources. In all our experiments so far, our estimates have been very accurate and differ by hardly one or two percent when compared to actual resource utilization. For BRAM, the estimate is always exact.

Packing the Most

In our tool, we first try to assign as many FSL channels to BRAM as possible. After all the BRAMs on the device are used, we assign them to LUT. BRAM implementation is faster to synthesize since only the access logic to the memory has to be synthesized, while in LUT implementation the whole FIFO is constructed using logic. It should be noted, however, that since BRAM implementation assigns the whole memory block in discrete amounts, it might be a waste of resources to assign the whole block when a FIFO of small depth is needed. Currently, this trade-off is not taken into account in our tool. Further, it might also be interesting to allow sharing of BRAM for multiple FIFOs. However, this requires extra control logic and arbitration to ensure fairness.

4 Experiments and Results

In this section, we present some of the results that were obtained by implementing several real and randomly-generated application SDF graphs. Here we show that our flow reduces the implementation gap between system level and RTL level design, and allows for more accurate performance evaluation using an emulation platform compared to simulation (Theelen et al. 2007) and analysis. Further, we see how our use-case partitioning approach minimizes the number of hardware designs by studying an example of applications running in a high-end mobile phone.

Our implementation platform is the Xilinx XUP Virtex II Pro Development Board with an *xc2vp30* FPGA on-board. Xilinx EDK 8.2i and ISE 8.2i were used for synthesis and implementation. All tools run on a Pentium 4 Core at 3 GHz with 1 GB of RAM.

Use-case Partitioning

In this section, we show the effectiveness of our approach to partition use-cases, and the heuristics to optimize on the execution time. This is demonstrated first using some random test cases and then with a case study involving applications in a mobile phone.

Using 10 random application SDF models, we generated all possible combinations giving a total of 1023 use-cases. We found that only 853 of these were feasible; the rest required more resources than were present on our FPGA device. In general, most use-cases of up to 6 applications could fit on the FPGA, while only a couple of use-cases with 7 applications were feasible.

When trying to compute partitions using the greedy method directly on these 853 use-cases, the algorithm terminated after 30 minutes without any result since there were too many sets to consider. When using the first-fit heuristic on these use-cases, we obtained a total of 145 partitions in 500 milli-seconds. However, since this approach is dependent on the order of use-cases, another order gave us a partition count of 126 in about 400 milli-seconds. After applying our reduction technique on feasible use-cases, 178 non-trivial use-cases were obtained. The greedy approach on these use-cases terminated in 3.3 seconds and resulted in 112 partitions. The first-fit heuristic on the non-trivial cases took 300 milli-seconds and gave 125 partitions, while another order of use-cases gave 116 partitions in about the same time.

A couple of observations can be made from this. Our techniques of use-case reduction are very effective in pruning the search space. Up to 80% of the use-cases are pruned away as trivial. This is essential in this case for example, when otherwise no results are obtained for greedy. We observe that while the first-fit heuristic is a lot faster, the results depend heavily on the order of input use-cases. However, if the search space is large, first-fit may be the only heuristic for obtaining results.

Table 6.2 Performance evaluation of heuristics used for use-case reduction and partitioning

	Random Graphs		Mobile Phone	
	# Partitions	Time (ms)	# Partitions	Time (ms)
Without Reduction				
Without Merging	853	–	23	–
Greedy	Out of Memory	–	Out of Memory	–
First-Fit	126	400	2	200
With Reduction				
Without Merging	178	100	3	40
Greedy	112	3300	2	180
First-Fit	116	300	2	180
Optimal Partitions	$\geq$110	–	2	–
Reduction Factor	7	–	11	–

Mobile-Phone Case Study

Here we consider 6 applications – video encoding (H263) (Hoes 2004), video decoding (Stuijk 2007), JPEG decoding (de Kock 2002), mp3 decoding, modem (Bhattacharyya et al. 1999), and a voice call. We first constructed all possible use-cases giving 63 use-cases in total. Some of these use-cases are not realistic, for example, JPEG decoding is unlikely to run together with video encoding or decoding, because when a person is recording or watching video, he/she will not be browsing the pictures. Similarly, listening to mp3 while talking on the phone is unrealistic. After pruning away such unrealistic use-cases we were left with 23 use-cases. After reduction to non-trivial use-cases, only 3 use-cases remained.

A greedy approach only works on the set after reduction. We observe that 23 use-cases is too much to handle if there are a lot of possible sub-sets. (In the previous example with 10 applications, we obtained 178 use-cases after reduction but since no partition was able to fit more than 4 use-cases, the total number of possible sets was limited.) After reduction, however, the greedy algorithm gives two partitions in 180 milli-seconds. The same results are obtained with the first-fit heuristic. However, the first-fit heuristic also solves the problem without pruning away the use-cases. Here the order only affects which trivial use-cases are attached to the non-trivial use-cases. In total, since we have only 2 partitions, performance of all the 23 use-cases is determined in about 2 hours. Without this reduction it would have taken close to 23 hours. Use-case merging and partitioning approach leads to a 11-fold reduction. The results are fed to the computer and stored on the CF card for later retrieval.

Table 6.2 shows how well our use-case reduction and partitioning heuristics perform. The time spent in corresponding steps is also shown. Reduction to non-trivial use-cases for mobile-phone case study takes 40 milli-seconds, for example, and

leaves us with only 3 use-cases. As mentioned earlier, the greedy heuristic for partitioning does not terminate with the available memory resources, when applied without reducing the use-cases. The design-space is too large to evaluate the largest feasible set. After reducing to non-feasible use-cases for random-graphs, we obtain 178 use-cases and at most 4 use-cases fit in any partition. Since the maximum error in using the greedy approach is given by $\ln|X| + 1$, where X is the number of elements in the largest partition, we get a maximum error of $\ln|4| + 1$ i.e. 2.38. We can therefore be sure that the minimum number of partitions is at least 110. We see a 7-fold reduction in the number of hardware configurations in the random-graphs use-case and about 11-fold in the mobile phone case study. Therefore, we can conclude that our heuristics of use-case reduction and partition are very effective in reducing the design time and in reducing the number of partitions.

Reconfiguration Time

The time to reconfigure an FPGA varies with the size of configuration file and the mode of reconfiguration. For Virtex II Pro 30, the configuration file is about 11 Mbits. The used CF card controller provides configuration bandwidth of 30 Mbit/s, translating to about 370 milli-seconds for reconfiguration. Configuring through the on-board programmable memory is a lot faster since it provides bandwidth of up to 800 Mbit/s. Thus, for the above FPGA device it takes only about 13 milli-seconds. The USB connection is a lot slower, and often takes about 8 seconds. However, for the end-design, we expect the configurations to be stored in a programmable memory on board that are retrieved as and when use-cases are enabled. A typical mobile-phone user is unlikely to start a new use-case more than once every few minutes. Therefore, the reconfiguration overhead of 13 milli-seconds is not significantly large, and is amortized over the duration of the use-case.

5 Suggested Readings

The multiple use-case concept is relatively new to the MPSoC, and one of the related research done is presented in (Murali et al. 2006). However, this focuses on supporting multiple use-cases for the communication infrastructure, in particular network-on-chip. Our flow is mainly targeted towards supporting multiple use-cases from computation perspective. In addition, we generate dedicated point-to-point connections for all the use-cases that are to be supported. It should be mentioned that links are shared across use-cases but not within the use-case.

Our definition of a *use-case* is similar to what is defined as a *scenario* in (Paul et al. 2006). The authors in (Paul et al. 2006) motivate the use of a scenario-oriented (or *use-case* in our paper) design flow for heterogeneous MPSoC platforms. Our approach provides one such design flow where designers can study the performance of all use-cases in an automated way and tune the architecture to achieve better performance of all applications in a use-case. The biggest advantage of our approach

is that we provide a real synthesized MPSoC platform for designers to play with and measure performance. Further, in (Paul et al. 2006) the architecture is provided as an input and is static, while we generate platforms given the application and use-case descriptions and the architecture is changed (reconfigured) dynamically for the different use-cases.

6 Conclusions

In this chapter, we propose a design-flow to generate architecture designs for multiple use-cases. Our approach takes the description of multiple use-cases and produces the corresponding MPSoC platform. A use-case is defined as a set of applications active concurrently. This is the first flow that allows mapping of multiple use-cases on a single platform. We propose techniques to merge and partition use-cases in order to minimize hardware requirements. The tool developed using this flow is made available online, and a stand-alone GUI tool is developed for both Windows and Linux. The flow allows designers to traverse the design space quickly, thus making the DSE of concurrently executing applications feasible. The heuristics used for use-case merging and partitioning reduce the design-exploration time 11-fold in a case study with mobile phone applications.

Further, we provide techniques to estimate resource utilization in an FPGA without carrying out the actual synthesis. This saves precious time during DSE and is very accurate as verified by the results. Our approach is also capable of minimizing the number of reconfigurations in the system. The use-case partitioning algorithm can be adapted to consider the relative frequency of the use of each use-case. The use-cases should be first sorted in the decreasing order of their use, and then the first-fit algorithm proposed in an earlier section should be applied. The algorithm will therefore first pack all the most frequently used use-cases together in one hardware partition, thereby reducing the reconfiguration from one frequently used use-case into another. However, for an optimal solution of the partition problem, many other parameters need to be taken into account, for example, reconfiguration time and average duration for each use-case. We would like to extend the use-case partitioning algorithm to take the exact reconfiguration overhead into account.

We would also like to develop and automate more ways of design space exploration, for example trying different mappings of applications. We would also like to try different kinds of arbiters in the design to improve fairness and allow for load-balancing between multiple applications.

Chapter 7
Conclusions and Open Problems

As we have seen so far in this book, a lot of progress has been made from both application and system-design perspective. While on one hand, this progress has enabled the designers to build better systems, on the other hand, it has also increased the expectations and demands of consumers. Consumers expect multimedia information to be available to them all the time. To satisfy their demand, the number of features in modern multimedia systems has been increasing. This has led to using multiprocessor systems for modern multimedia systems. In this chapter, the major conclusions about designing multimedia multiprocessor systems are presented, together with several issues that remain to be solved.

1 Conclusions

The design of multimedia platforms is becoming increasingly more complex. Modern multimedia systems need to support a large number of applications or functions in a single device. To achieve high performance in such systems, more and more processors are being integrated into a single chip to build Multi-Processor Systems-on-Chip (MPSoCs). The heterogeneity of such systems is also increasing with the use of specialized digital hardware, application domain processors and other IP (intellectual property) blocks on a single chip, since various standards and algorithms are to be supported. These embedded systems also need to meet timing and other *non-functional* constraints like low power (low-power favours heterogeneity) and design area. Further, processors designed for multimedia applications (also known as *streaming processors*) often do not support preemption to keep costs low, making traditional analysis techniques unusable.

To achieve high performance in such systems, the limited computational resources must be shared. The concurrent execution of dynamic applications on shared resources causes interference. The fact that these applications do not always run concurrently adds a new dimension to the design problem. We defined each such combination of applications executing concurrently as a *use-case*. Currently, companies often spend 60–70% of the product development cost in verifying all feasible

A. Kumar et al., *Multimedia Multiprocessor Systems*, Embedded Systems,
DOI 10.1007/978-94-007-0083-3_7,

use-cases. Having an efficient, but accurate analysis technique can significantly reduce this development cost. Since applications are often added to the system at run-time (for example, a mobile-phone user may download a Java application at run-time), a complete analysis at design-time is also not feasible. Existing techniques are unable to handle this dynamism, and the only solution left to the designer is to over-dimension the hardware by a large factor leading to increased area, cost and power.

In Chap. 3 of this book, a run-time performance prediction methodology is presented that can accurately and quickly predict the performance of multiple applications before they execute in the system. Synchronous data flow (SDF) graphs are used to model applications, since they fit well with characteristics of multimedia applications, and at the same time allow analysis of application performance. Further, their atomic execution requirement matches well with the non-preemptive nature of many streaming processors. While a lot of techniques are available to analyze performance of single applications, for multiple applications this task is a lot harder and little work has been done in this direction. This book presents one of the first attempts to analyze performance of multiple applications executing on heterogeneous non-preemptive multiprocessor platforms.

Our technique uses performance expressions computed off-line from the application specifications. A run-time iterative probabilistic analysis is used to estimate the time spent by tasks during the contention phase, and thereby predict the performance of applications. The average error in prediction using iterative probability turns out to be only a few percent. Further, it takes about four to six iterations for the prediction to converge. The complexity and execution time of the algorithm is very low – it takes only 3 ms to evaluate the performance of ten applications on a 50 MHz embedded processor. This also proves the suitability of the technique for design space exploration on a regular desktop running at about 3 GHz where the same analysis takes just 50 microseconds.

Further, we presented a design-flow for designing systems with multiple applications in Chap. 4. A hybrid approach is presented where the time-consuming application-specific computations are done at design-time, and in isolation from other applications, and the use-case-specific computations are performed at run-time. This allows easy addition of applications at run-time. Further, a run-time mechanism is presented to manage resources in a system. This ensures that no application starves due to another application. This mechanism enforces budgets and suspends applications if they achieve a higher performance than desired. This allows other applications to also achieve their desired performance. A resource manager (RM) is presented to manage computation and communication resources, and to achieve the above goals of performance prediction, admission control and budget enforcement. A case-study done with two application models – H263 and JPEG, demonstrates the effectiveness of budget enforcement in achieving the desired performance of both applications.

With higher consumer demands the time-to-market has become significantly lower. To cope with the complexity in designing such systems, a largely automated design-flow is needed that can generate systems from a high-level architectural description such that they are not error-prone and their design consumes less time.

A highly automated flow – *MAMPS* (Multi-Application Multi-Processor Synthesis) is presented in Chap. 5, that synthesizes multi-processor platforms for multiple applications specified in the form of SDF graph models. The flow has been used to implement a tool that directly generates multi-processor designs for Xilinx FPGAs, complete with hardware and software descriptions. A case study done with the tool shows the effectiveness of the tool in which 24 design points were explored to compute the optimal buffer requirements of multiple applications in about 45 minutes including FPGA synthesis time.

One of the key design automation challenges that remain is fast exploration of software and hardware implementation alternatives with accurate performance evaluation, also known as design space exploration (DSE). A design methodology is presented in Chap. 6 to generate multiprocessor systems in a systematic and fully automated way for *multiple use-cases*. Techniques are presented to merge multiple use-cases into one hardware design to minimize cost and design time, making it well-suited for fast DSE of MPSoC systems. Heuristics to partition use-cases are also presented such that each partition can fit in an FPGA, and all use-cases can be catered for. A case study with mobile-phone applications shows an 11-fold reduction in DSE time.

2 Open Problems

While this book presents solutions to various problems in analysis, design and management of multimedia multiprocessor systems, a number of issues remain to be solved. Some of these are listed below.

(1) **Hard-real time support:** While the analysis techniques presented in this book are aimed towards multimedia systems that do not require a hard-bound on performance, they can easily be extended to support hard-real time applications as well. However, as has been mentioned earlier, that generally translates to a poor resource utilization. Techniques that can achieve high utilization and provide hard-bounds on performance still need to be developed. One option is to consider joining multiple application graphs with very few edges – only the minimum number needed to achieve rate-control – and then derive a static-order for that graph. This would achieve high-utilization and provide hard-bounds on performance. However, a potential drawback of this scheme is that for every possible use-case, a static order has to be stored, and care has to be taken that the system does not go into deadlock while switching between use-cases.

(2) **Soft-real time guarantee:** The analysis technique presented in Chap. 3 is very accurate and fast. However, it does not provide any guarantee on the accuracy of the results. Even for soft-real time applications like video and audio processing, some sort of measure of accuracy of results is desirable. Extending the probabilistic analysis to support this would increase the potential of this analysis technique. The designer wants to know how the applications would perform with the given resources, and accordingly increase or decrease the resources needed for system design.

(3) **Model network and memory:** In this book, we have often ignored the contention for memory and network resources. Even though, in theory (Stuijk 2007), this contention can be naturally modeled in SDF graph as well, it remains to be seen how well the technique applies, and how does it affect the performance of multiple applications. With a complete design-flow where the memory and network contention are also modeled, a designer will be able to make choices about the distribution of memory and network resources in the system, and about the allocation to different applications.

(4) **SDF derivation:** Throughout this book, we have assumed that an SDF model of application has already been derived. In practice, this task can be very time consuming, and is mostly manual. Automated extraction of parallel models from a sequential specification of an application is still an open problem. While a lot of tools are available to help in this derivation, most of them require extensive human interaction (Cockx et al. 2007). This makes the design space exploration very time-consuming. The extraction of worst-case execution-times needed for each task is also very difficult. While static code analysis can provide very poor estimates for task execution-time, profiling is only as accurate as the input sequence. This makes the compromise between an accurate and reasonable model rather difficult.

(5) **Other models:** In this book, we have used synchronous dataflow for modeling applications. While these models are very good in expressing streaming behaviour, they are not always the best for expressing the dynamism in the applications. A number of other models are available that allow for dynamic behaviour in the model, e.g. CSDF (Lauwereins et al. 1994; Bilsen et al. 1996), SADF (Theelen et al. 2006) and KPN (Kahn 1974). While CSDF is still static, it allows for periodically changing channel rates during different iterations of the actors. Developing analysis techniques for those models would help provide predictability to dynamic applications as well, and satisfy both the designers and the consumers. While extension to CSDF is rather trivial, it only provides a limited solution. Modeling applications in SADF is a very promising solution since it can capture dynamism and still provide guarantees about application behaviour. The latter is unfortunately, not possible when KPN is used to model applications.

(6) **Achieving predictability in suspension:** In Sect. 3 of Chap. 4, a technique has been suggested to achieve predictability by using suspension. This technique is very powerful as it allows the designer to specify the desired application performance. By varying the time the system spends in each state, the performance of applications can be changed. While the basic idea has been outlined, the size of the time-wheel affects the performance significantly. The technique can also be adapted to support hard-real time tasks by using a conservative analysis, such as worst-case waiting-time analysis.

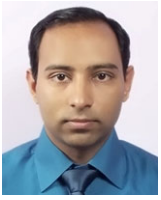
(7) **Design space exploration heuristics:** In this book, we have concentrated on enabling design space exploration. Various techniques are provided for performance analysis of multiple applications to give feedback to the designer. Further, hardware design flow for rapid prototyping and performance evaluation

is provided. However, we have not focused on heuristics to explore mapping options for optimizing performance and generating designs that satisfy the constraints of area and power.

(8) **Optimizing the use-case partitions:** The use-case partitioning algorithm can be adapted to consider the relative frequency of the use of each use-case. The use-cases can first be sorted in the decreasing order of their use, and then the first-fit algorithm can be applied. The algorithm can therefore first pack all the most frequently used use-cases together in one hardware partition, thereby reducing the reconfiguration from one frequently used use-case into another. However, for an optimal solution of the partition problem, many other parameters need to be taken into account, for example reconfiguration time and average duration for each use-case. More research needs to be done in this to verify the suitability and effectiveness of this approach.

The above are some of the issues that need to be solved to take the analysis, design and management of multimedia multiprocessor systems into the next era. We hope that this book was not only enjoyed by the readers, but also interested many researchers to solve the above challenges.

About the Authors

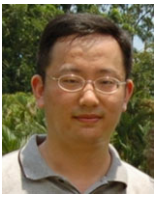


Akash Kumar was born in Bijnor, India on November 13, 1980. After finishing the middle high-school at the Dayawati Modi Academy in Rampur, India in 1996, he proceeded to Raffles Junior College, Singapore for his pre-university education. In 2002, he completed Bachelors in Computer Engineering (First Class Honours) from the National University of Singapore (NUS), and in 2004 he completed joint Masters in Technological Design (Embedded Systems) from Eindhoven University of Technology (TUe) and NUS. In 2005, he began working towards his joint Ph.D. degree from TUe and NUS in the Electronic Systems group and Electrical and Computer Engineering department respectively. His research was funded by STW within the PreMaDoNA project. It has led, among others, to several publications and this book. Presently, Akash is a visiting fellow at the Department of Electrical and Computer Engineering, NUS, Singapore. His research interests include analysis, design methodologies, and resource management of multi-processor systems.

Henk Corporaal has gained a M.Sc. in Theoretical Physics from the University of Groningen, and a Ph.D. in Electrical Engineering, in the area of Computer Architecture, from Delft University of Technology.
Corporaal has been teaching at several schools for higher education, has been associate professor at the Delft University of Technology in the field of computer architecture and code generation, had a joint professor appointment at the National University of Singapore, and has been scientific director of the joined NUS-TUE Design Technology Institute. He also has been department head and chief scientist within the DESICS (Design Technology for Integrated Information and Communication Systems) division at IMEC, Leuven (Belgium).
Currently Corporaal is Professor in Embedded System Architectures at the Eindhoven University of Technology (TU/e) in The Netherlands. He has co-authored over 250 journal and conference papers in the (multi-)processor architecture and embedded system design area. Furthermore he invented a new class of VLIW architectures, the Transport Triggered Architectures, which is used in several commercial products, and by many research groups. His current research projects are on the predictable design of soft- and hard real-time embedded systems.

A. Kumar et al., *Multimedia Multiprocessor Systems*, Embedded Systems,
DOI 10.1007/978-94-007-0083-3,

Bart Mesman obtained his Ph.D. from the Eindhoven University of Technology in 2001. His thesis discusses an efficient constraint-satisfaction method for scheduling operations on a distributed VLIW processor architecture with highly constrained register files with stringent timing requirements. He has worked at Philips Research from 1995–2005 on DSP processor architectures and compilation. Dr. Mesman is currently employed by Eindhoven University. His research interests include (multi-)processor architectures, compile-time and run-time scheduling, and resource management in multi-media devices.



Yajun Ha received the B.S. degree in Electrical Engineering from Zhejiang University, Hangzhou, China, in 1996, the M.Eng. degree in Electrical Engineering from the National University of Singapore (NUS), Singapore, in 1999, and the Ph.D. degree in Electrical Engineering from Katholieke Universiteit Leuven, Leuven, Belgium, in 2004. He has been an assistant professor at the Department of Electrical and Computer Engineering, NUS, since 2004. Between 1999 and 2004, he did his Ph.D. research project at IMEC, Leuven. His research interests lie in the embedded system architecture and design methodologies, particularly in the area of reconfigurable computing. He has held a US patent and published more than 50 internationally refereed technical papers in his interested areas.

Glossary

ASIC Application specific integrated circuit
ASIP Application specific instruction-set processor
BDF Boolean dataflow
CF Compact flash
CSDF Cyclo static dataflow
DCT Discrete cosine transform
DSE Design space exploration
DSP Digital signal processing
FCFS First-come-first-serve
FIFO First-in-first-out
FPGA Field-programmable gate array
FSL Fast simplex link
HSDFG Homogeneous synchronous dataflow graph
IDCT Inverse discrete cosine transform
IP Intellectual property
JPEG Joint Photographers Expert Group
KPN Kahn process network
LUT Lookup table
MAMPS Multi-Application Multi-Processor Synthesis
MB Microblaze
MoC Models of Computation
MCM Maximum cycle mean
MPSoC Multi-processor system-on-chip
POOSL Parallel object oriented specification language
QoS Quality-of-service
RAM Random access memory
RCSP Rate controlled static priority
RISC Reduced instruction set computing
RM Resource manager
RR Round-robin
RRWS Round-robin with skipping

A. Kumar et al., *Multimedia Multiprocessor Systems*, Embedded Systems,
DOI 10.1007/978-94-007-0083-3,

RTL Register transfer level
SADF Scenario aware dataflow
SDF Synchronous dataflow
SDFG Synchronous dataflow graph
SMS Short messaging service
TDMA Time-division multiple access
VLC Variable length coding
VLD Variable length decoding
VLIW Very long instruction word
WCET Worst case execution time
WCRT Worst case response time
XML Extensible markup language

Actor A program segment of an application modeled as a vertex of a graph that should be executed atomically.

Composability Mapping and analysis of performance of multiple applications on a multiprocessor platform in isolation.

Control token Some information that controls the behaviour of actor. It can determine the rate of different ports in some MoC (like SADF and BDF), and the execution time in some other MoC (like SADF and KPN).

Critical instant The critical instant for an actor is defined as an instant at which a request for that actor has the largest response time.

Multimedia systems Systems that use a combination of content forms like text, audio, video, pictures and animation to provide information or entertainment to the user.

Output actor The last task in the execution of an application after whose execution one iteration of the application can be said to have been completed.

Rate The number of tokens that need to be consumed (for input rate) or produced (for output rate) during an execution of an actor.

Reconfigurable platform A piece of hardware that can be programmed or reconfigured at run-time to achieve the desired functionality.

Response time The time an actor takes to respond once it is ready i.e. the sum of its waiting and its execution time.

Scenario A mode of operation of a particular application. For example, an MPEG video stream may be decoding an I-frame or a B-frame or a P-frame. The resource requirement in each scenario may be very different.

Scheduling Process of determining when and where a part of an application is to be executed.

Task A program segment of an application that is executed atomically.

Token A data element that is consumed or produced during an actor-execution.

Use-case This refers to a combination of applications that may be active concurrently. Each such combination is a new use-case.

Work-conserving schedule This implies if there is work to be done (or task to be executed) on a processor, it will execute it and not wait for some other work (or task). A schedule is work-conserving when the processor is not idle as long as there is any task waiting and ready to execute on the processor.

References

Abeni, L., Buttazzo, G.: QoS guarantee using probabilistic deadlines. In: Proceedings of the 11th Euromicro Conference on Real-Time Systems, 1999, pp. 242–249. IEEE, York (1999). doi:10.1109/EMRTS.1999.777471

Adee, S.: The data: 37 years of Moore's law. IEEE Spectr. **45**(5), 56 (2008). doi:10.1109/MSPEC.2008.4505312

Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: AFIPS '67 (Spring): Proceedings of the Spring Joint Computer Conference, April 18–20, 1967, pp. 483–485. ACM, New York (1967). doi:10.1145/1465482.1465560

Artieri, A., Alto, V.D., Chesson, R., Hopkins, M., Rossi, M.C.: Nomadik open multimedia platform for next-generation mobile devices. Technical Article TA305, ST Microelectronics (2003). www.st.com

Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., et al.: The landscape of parallel computing research: a view from Berkeley. Technical Report 2006-183, Electrical Engineering and Computer Sciences, University of California Berkeley (2006)

Baldwin, T.F., McVoy, D.S., Steinfield, C.: Convergence: Integrating Media, Information & Communication. Sage, Thousand Oaks (1996)

Bambha, N., Kianzad, V., Khandelia, M., Bhattacharyya, S.S.: Intermediate representations for design automation of multiprocessor DSP systems. Des. Autom. Embed. Syst. **7**(4), 307–323 (2002). doi:10.1023/A:1020307222052

Baruah, S.: The non-preemptive scheduling of periodic tasks upon multiprocessors. Real-Time Syst. **32**(1), 9–20 (2006). doi:10.1007/s11241-006-4961-9

Baruah, S.K., Cohen, N.K., Plaxton, C.G., Varvel, D.A.: Proportionate progress: a notion of fairness in resource allocation. Algorithmica **15**, 600–625 (1996)

Bekooij, M., Hoes, R., Moreira, O., Poplavko, P., Pastrnak, M., Mesman, B., Mol, J.D., Stuijk, S., Gheorghita, V., van Meerbergen, J.: Dataflow analysis for real-time embedded multiprocessor system design. In: Dynamic and Robust Streaming in and Between Connected Consumer-Electronic Devices, pp. 81–108. Springer, Berlin (2005)

Bertozzi, S., Acquaviva, A., Bertozzi, D., Poggiali, A.: Supporting task migration in multiprocessor systems-on-chip: a feasibility study. In: DATE '06: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 15–20. European Design and Automation Association, Leuven (2006)

Bhattacharyya, S.S., Murthy, P.K., Lee, E.A.: Synthesis of embedded software from synchronous dataflow specifications. VLSI Signal Process. **21**(2), 151–166 (1999)

Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.: Cycle-static dataflow. IEEE Trans. Signal Process. **44**(2), 397–408 (1996). doi:10.1109/78.485935

Bluetooth SIG: Bluetooth specification version 2.0+ EDR. Bluetooth SIG Standard (2004)

A. Kumar et al., *Multimedia Multiprocessor Systems*, Embedded Systems,
DOI 10.1007/978-94-007-0083-3, © Springer Science+Business Media B.V. 2011

Borkar, S.: Thousand core chips: a technology perspective. In: DAC '07: Proceedings of the 44th Annual Conference on Design Automation, pp. 746–749. ACM, New York (2007). doi:10.1145/1278480.1278667

Buck, J.T., Lee, E.A.: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In: ICASSP-93, 1993 IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 1, pp. 429–432 (1993). doi:10.1109/ICASSP.1993.319147

Cai, Y., Kong, M.C.: Nonpreemptive scheduling of periodic tasks in uni- and multiprocessor systems. Algorithmica **15**(6), 572–599 (1996)

CES: Consumer electronics show (2009). Available from: http://www.cesweb.org/

Cockx, J., Denolf, K., Vanhoof, B., Stahl, R.: Sprint: a tool to generate concurrent transaction-level models from sequential code. EURASIP J. Appl. Signal Process. **2007**(1), 213 (2007). doi:10.1155/2007/75373

Coffman Jr., E.G., Muntz, R.R., Trotter, H.: Waiting time distributions for processor-sharing systems. J. ACM **17**(1), 123–130 (1970). doi:10.1145/321556.321568

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, second edn., pp. 1033–1038. MIT Press, Cambridge (2001)

Cumming, P.: The TI OMAP platform approach to Soc. In: Winning the SoC Revolution. Kluwer Academic, Dordrecht (2003)

Dasdan, A.: Experimental analysis of the fastest optimum cycle ratio and mean algorithms. ACM Trans. Des. Autom. Electron. Syst. **9**(4), 385–418 (2004). doi:10.1145/1027084.1027085

Davari, S., Dhall, S.K.: An on line algorithm for real-time tasks allocation. In: IEEE Real-Time Systems Symposium, pp. 194–200 (1986)

de Kock, E.A.: Multiprocessor mapping of process networks: a JPEG decoding case study. In: Proceedings of 15th ISSS, pp. 68–73. IEEE Computer Society, Los Alamitos (2002)

Denolf, K., Bekooij, M., Cockx, J., Verkest, D., Corporaal, H.: Exploiting the expressiveness of cyclo-static dataflow to model multimedia implementations. EURASIP J. Adv. Signal Process. (2007). doi:10.1155/2007/84078

De Oliveira, J.A., Van Antwerpen, H.: The Philips Nexperia digital video platforms. In: Winning the SoC Revolution. Kluwer Academic, Dordrecht (2003)

Gao, G.R.: A pipelined code mapping scheme for static data flow computers. Ph.D. thesis, Massachusetts Institute of Technology (1983)

Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-completeness. Freeman, New York (1979)

Ghamarian, A.H.: Timing analysis of synchronous data flow graphs. Ph.D. thesis, Eindhoven University of Technology (2008)

Ghamarian, A.H., Geilen, M.C.W., Stuijk, S., Basten, T., Theelen, B.D., Mousavi, M.R., Moonen, A.J.M., Bekooij, M.J.G.: Throughput analysis of synchronous data flow graphs. In: Sixth International Conference on Application of Concurrency to System Design (ACSD), pp. 25–36. IEEE Computer Society, Los Alamitos (2006). doi:10.1109/ACSD.2006.33

Ghamarian, A.H., Geilen, M.C.W., Basten, T., Stuijk, S.: Parametric throughput analysis of synchronous data flow graphs. In: Design Automation and Test in Europe, pp. 116–121. IEEE Computer Society, Los Alamitos (2008). doi:10.1109/DATE.2008.4484672

Gonzalez, R.E.: Xtensa: a configurable and extensible processor. IEEE MICRO **20**(2), 60–70 (2000). doi:10.1109/40.848473

Goossens, K., Dielissen, J., Radulescu, A.: Æthereal network on chip: concepts, architectures, and implementations. IEEE Des. Test Comput. **22**(5), 414–421 (2005). doi:10.1109/MDT.2005.99

Halfhill, T.R.: Busy bees at Silicon Hive. Microprocessor Report (2005)

Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. Computer **41**(7), 33–38 (2008). doi:10.1109/MC.2008.209

Hive, Silicon: Silicon hive (2004). Available from: http://www.siliconhive.com

Hoes, R.: Predictable dynamic behavior in NoC-based MPSoC (2004). Available from: www.es.ele.tue.nl/epicurus/

HOL: Head-of-line blocking [online] (2009). Available from: http://en.wikipedia.org/wiki/Head-of-line_blocking

Hua, S., Qu, G., Bhattacharyya, S.S.: Probabilistic design of multimedia embedded systems. ACM Trans. Embed. Comput. Syst. **6**(3), 15 (2007). doi:10.1145/1275986.1275987

Intel: Intel press kit (2004). Available from: http://www.intel.com/pressroom/kits/45nm/index.htm

Jeffay, K., Stanat, D.F., Martel, C.U.: On non-preemptive scheduling of periodic and sporadic tasks. In: Proceedings of 12th IEEE Real-Time Systems Symposium, pp. 129–139 (1991)

Jerraya, A., Bacivarov, I.: Performance evaluation methods for multiprocessor system-on-chip design. In: EDA for IC System Design, Verification and Testing, pp. 6.1–6.14. Taylor and Francis, London (2006)

Jerraya, A., Wolf, W.: Multiprocessor Systems-on-Chips. Morgan Kaufmann, San Francisco (2004)

Jin, Y., Satish, N., Ravindran, K., Keutzer, K.: An automated exploration framework for FPGA-based soft multiprocessor systems. In: 3rd CODES+ISSS, pp. 273–278. IEEE Computer Society, Los Alamitos (2005). doi:10.1145/1084834.1084903

Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the cell multiprocessor. IBM Corporation (2005)

Kahn, G.: The semantics of a simple language for parallel programming. Inf. Process. **74**, 471–475 (1974)

Kahn, J.M., Barry, J.R.: Wireless infrared communications. Proc. IEEE **85**(2), 265–298 (1997)

Karp, R.M., Miller, R.E.: Properties of a model for parallel computations: determinancy, termination, queueing. SIAM J. Appl. Math. **14**(6), 1390–1411 (1966)

Keutzer, K., Malik, S., Newton, A.R., Rabaey, J.M., Sangiovanni-Vincentelli, A.: System-level design: orthogonalization of concerns and platform-based design. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **19**(12), 1523–1543 (2000)

Kopetz, H., Obermaisser, R.: Temporal composability [real-time embedded systems]. Comput. Control Eng. J. **13**(4), 156–162 (2002)

Kopetz, H., Suri, N.: Compositional design of RT systems: a conceptual basis for specification of linking interfaces. In: ISORC '03: Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, p. 51. IEEE Computer Society, Washington (2003). doi:10.1109/ISORC.2003.1199236

Kumar, A., Mesman, B., Theelen, B., Corporaal, H., Yajun, H.: Resource manager for non-preemptive heterogeneous multiprocessor system-on-chip. In: ESTMED '06: Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia, pp. 33–38. IEEE Computer Society, Washington (2006a). doi:10.1109/ESTMED.2006.321271

Kumar, A., Mesman, B., Corporaal, H., van Meerbergen, J., Yajun, H.: Global analysis of resource arbitration for MPSoC. In: DSD '06: Proceedings of the 9th EUROMICRO Conference on Digital System Design, pp. 71–78. IEEE Computer Society, Washington (2006b). doi:10.1109/DSD.2006.57

Kumar, A., Hansson, A., Huisken, J., Corporaal, H.: An FPGA design flow for reconfigurable network-based multi-processor systems on chip. In: Design, Automation and Test in Europe, pp. 117–122. IEEE Computer Society, Los Alamitos (2007)

Kumar, A., Fernando, S., Ha, Y., Mesman, B., Corporaal, H.: Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA. ACM Trans. Des. Autom. Electron. Syst. **13**(3), 1–27 (2008). doi:10.1145/1367045.1367049

Kumar, A., Mesman, B., Corporaal, H., Ha, Y.: Iterative probabilistic performance prediction for multi-application multiprocessor systems. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **29**(4), 538–551 (2010). doi:10.1109/TCAD.2010.2042887

Kunzli, S., Poletti, F., Benini, L., Thiele, L.: Combining simulation and formal methods for system-level performance analysis. In: Design, Automation and Test in Europe, vol. 1, pp. 1–6. IEEE Computer Society, Los Alamitos (2006)

Lauwereins, R., Wauters, P., Ade, M., Peperstraete, J.A.: Geometric parallelism and cyclo-static data flow in grape-ii. In: Fifth International Workshop on Rapid System Prototyping, 1994, Shortening the Path from Specification to Prototype, Proceedings, pp. 90–107 (1994). doi:10.1109/IWRSP.1994.315905

Lauwereins, R., Wong, C., Marchal, P., Vounckx, J., David, P., Himpe, S., Catthoor, F., Yang, P.: Managing dynamic concurrent tasks in embedded real-time multimedia systems. In: Proceedings of the 15th International Symposium on System Synthesis, pp. 112–119. IEEE Computer Society, Los Alamitos (2002). doi:10.1109/ISSS.2002.1227162

Lee, E.A.: Consistency in dataflow graphs. IEEE Trans. Parallel Distrib. Syst. **2**(2), 223–235 (1991). doi:10.1109/71.89067

Lee, E.A., Ha, S.: Scheduling strategies for multiprocessor real-time DSP. In: GLOBECOM '89, Global Telecommunications Conference and Exhibition, Communications Technology for the 1990s and Beyond, vol. 2, pp. 1279–1283 (1989). doi:10.1109/GLOCOM.1989.64160

Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous dataflow programs for digital signal processing. IEEE Trans. Comput. **36**(1), 24–35 (1987)

Leontyev, H., Chakraborty, S., Anderson, J.H.: Multiprocessor extensions to real-time calculus. In: Proceedings of the 30th IEEE Real-Time Systems Symposium, pp. 410–421. IEEE Computer Society, Los Alamitos (2009). doi:10.1109/RTSS.2009.29

Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM **20**(1), 46–61 (1973). doi:10.1145/321738.321743

Lyonnard, D., Yoo, S., Baghdadi, A., Jerraya, A.A.: Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In: Design Automation Conference, pp. 518–523. ACM, New York (2001)

Magoon, R., Molnar, A., Zachan, J., Hatcher, G., Rhee, W., Inc, S.S., Beach, N.: A single-chip quad-band (850/900/1800/1900 MHz) direct conversion GSM/GPRS RF transceiver with integrated VCOs and fractional-n synthesizer. IEEE J. Solid-State Circuits **37**(12), 1710–1720 (2002)

MAMPS: Multi-application multi-processor synthesis [online] (2009). Username: `todaes`, password: `guest`. Available at: http://www.es.ele.tue.nl/mamps/

Manolache, S., Eles, P., Peng, Z.: Schedulability analysis of applications with stochastic task execution times. ACM Trans. Embed. Comput. Syst. **3**(4), 706–735 (2004). doi:10.1145/1027794.1027797

Masrur, A., Chakraborty, S., Farber, G.: Constant-time admission control for deadline monotonic tasks. In: Proceedings of the IEEE DATE, pp. 220–225 (2010)

Moore, G.E.: Cramming more components onto integrated circuits. Electron. Mag. **38**(8), 114–117 (1965)

Moreira, O., Mol, J.J.-D., Bekooij, M.: Online resource management in a multiprocessor with a network-on-chip. In: SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing, pp. 1557–1564. ACM, New York (2007). doi:10.1145/1244002.1244335

Murali, S., Coenen, M., Radulescu, A., Goossens, K., De Micheli, G.: A methodology for mapping multiple use-cases onto networks on chips. In: Design, Automation and Test in Europe, pp. 118–123. IEEE Computer Society, Los Alamitos (2006)

Murata, T.: Petri nets: properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989). doi:10.1109/5.24143

Nikolov, H., Stefanov, T., Deprettere, E.: Multi-processor system design with ESPAM. In: Proceedings of the 4th CODES+ISSS, pp. 211–216. ACM, New York (2006)

Nikolov, H., Stefanov, T., Deprettere, E.: Systematic and automated multiprocessor system design, programming, and implementation. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **27**(3), 542–555 (2008). doi:10.1109/TCAD.2007.911337

Nollet, V.: Run-time management for future MPSoC platforms. Ph.D. thesis, Eindhoven University of Technology (2008)

Nollet, V., Avasare, P., Mignolet, J.-Y., Verkest, D.: Low cost task migration initiation in a heterogeneous mp-soc. In: Design, Automation and Test in Europe, pp. 252–253. IEEE Computer Society, Los Alamitos (2005)

Nollet, V., Avasare, P., Eeckhaut, H., Verkest, D., Corporaal, H.: Run-time management of a mpsoc containing fpga fabric tiles. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **16**(1), 24–33 (2008). doi:10.1109/TVLSI.2007.912097

Odyssey, Magnavox: World's first video game console (1972). Available from: http://en.wikipedia.org/wiki/Magnavox_Odyssey

Oh, H., Ha, S.: Fractional rate dataflow model for efficient code synthesis. J. VLSI Signal Process. **37**(1), 41–51 (2004)

Patterson, D.A., Ditzel, D.R.: The case for the reduced instruction set computer. Comput. Archit. News **8**(6), 25–33 (1980). doi:10.1145/641914.641917

Paul, J.M., Thomas, D.E., Bobrek, A.: Scenario-oriented design for single-chip heterogeneous multiprocessors. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **14**(8), 868–880 (2006). doi:10.1109/TVLSI.2006.878474

Petri, C.A.: Kommunikation mit Automaten. Ph.D. thesis, Rheinisch-Westfälisches Institut f. instrumentelle Mathematik an d. Univ. (1962)

Philips: Royal Philips (2009). Available from: www.philips.com

Pino, J.L., Lee, E.A.: Hierarchical static scheduling of dataflow graphs onto multiple processors. In: ICASSP-95, 1995 International Conference on Acoustics, Speech, and Signal Processing, vol. 4, pp. 2643–2646. IEEE, Detroit (1995). doi:10.1109/ICASSP.1995.480104

PS3: Sony PlayStation 3 (2006). Available from: http://www.playstation.com/

Richter, K., Jersak, M., Ernst, R.: A formal approach to MPSoC performance verification. Computer **36**(4), 60–67 (2003)

Robertazzi, T.G.: Computer Networks and Systems: Queueing Theory and Performance Evaluation. Springer, Berlin (2000)

Ross, P.E.: Why cpu frequency stalled. IEEE Spectr. **45**(4), 72 (2008). doi:10.1109/MSPEC.2008.4476447

Roza, E.: Systems-on-chip: what are the limits? Electron. Commun. Eng. J. **13**(6), 249–255 (2001)

Rul, S., Vandierendonck, H., De Bosschere, K.: Function level parallelism driven by data dependencies. Comput. Archit. News **35**(1), 55–62 (2007). doi:10.1145/1241601.1241612

Samsung: Samsung (2009). Available from: http://www.samsung.com

Sangiovanni-Vincentelli, A., Carloni, L., De Bernardinis, F., Sgroi, M.: Benefits and challenges for platform-based design. In: DAC '04: Proceedings of the 41st Annual Conference on Design Automation, pp. 409–414. ACM, New York (2004). doi:10.1145/996566.996684

Schliecker, S., Ernst, R.: A recursive approach to end-to-end path latency computation in heterogeneous multiprocessor systems. In: Proceedings of the 7th CODES+ISSS, pp. 433–442. ACM, New York (2009). doi:10.1145/1629435.1629494

SDF3: SDF for free [online] (2009). Available at: http://www.es.ele.tue.nl/sdf3/

Shabbir, A., Kumar, A., Mesman, B., Corporaal, H.: Enabling mpsoc design space exploration on fpgas. In: Proceedings of International Multi Topic Conference (IMTIC). Springer, New York (2008)

Shabbir, A., Kumar, A., Mesman, B., Corporaal, H.: Enabling MPSoC Design Space Exploration on FPGAs. Communications in Computer and Information Science, vol. 20, pp. 412–421 (Chap. 44). Springer, Berlin (2009). doi:10.1007/978-3-540-89853-5_44

Shabbir, A., Kumar, A., Stuijk, S., Mesman, B., Corporaal, H.: CA-MPSoC: an automated design flow for predictable multi-processor architectures for multiple applications. J. Systems Archit. (2010). doi:10.1016/j.sysarc.2010.03.007

Sony: World of Sony (2009). Available from: http://www.sony.com

Sriram, S., Bhattacharyya, S.S.: Embedded Multiprocessors; Scheduling and Synchronization. Dekker, New York (2000)

Stefanov, T., Zissulescu, C., Turjan, A., Kienhuis, B., Deprette, E.: System design using Kahn process networks: the Compaan/Laura approach. In: Design, Automation and Test in Europe, pp. 340–345. IEEE Computer Society, Los Alamitos (2004)

Stuijk, S.: Predictable mapping of streaming applications on multiprocessors. Ph.D. thesis, Eindhoven University of Technology (2007)

Stuijk, S., Geilen, M., Basten, T.: SDF3: SDF for free. In: Sixth International Conference on Application of Concurrency to System Design (ACSD), pp. 276–278. IEEE Computer Society, Los Alamitos (2006a). doi:10.1109/ACSD.2006.23

Stuijk, S., Geilen, M.C.W., Basten, T.: Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In: Design Automation Conference, pp. 899–904. ACM, New York (2006b)

Takacs, L.: Introduction to the Theory of Queues. Greenwood Press, New York (1962)
Tensilica: Tensilica – the dataplane processor company (2009). Available from: http://www.tensilica.com
Terr, D., Weisstein, E.W.: Symmetric polynomial (2008). Available from: mathworld.wolfram.com/SymmetricPolynomial.html
Teruel, E., Chrzastowski-Wachtel, P., Colom, J., Silva, M.: On weighted t-systems. In: Application and Theory of Petri Nets 1992, pp. 348–367 (1992). doi:10.1007/3-540-55676-1_20
Texas Instruments: TI resource page [online] (2008). Available from: http://www.ti.com
Theelen, B.D., Geilen, M.C.W., Basten, T., Voeten, J.P.M., Gheorghita, S.V., Stuijk, S.: A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In: Proceedings of the International Conference on Formal Methods and Models for Co-design. IEEE Computer Society Press, Los Alamitos (2006)
Theelen, B.D., Florescu, O., Geilen, M.C.W., Huang, J., van der Putten, P.H.A., Voeten, J.P.M.: Software/hardware engineering with the parallel object-oriented specification language. In: Proceedings of the Fifth ACM-IEEE International Conference on Formal Methods and Models for Codesign, pp. 139–148. IEEE Computer Society, Los Alamitos (2007)
Thiele, L., Chakraborty, S., Naedele, M.: Real-time calculus for scheduling hard real-time systems. In: Proceedings of ISCAS 2000, Geneva, vol. 4, pp. 101–104. IEEE, Geneva (2000). doi:10.1109/ISCAS.2000.858698
TNS: TNS research [online] (2006). Available from: http://www.tns.lv/?lang=en&category=showuid&id=2288
Wawrzynek, J., Patterson, D., Oskin, M., Lu, S.L., Kozyrakis, C., Hoe, J.C., Chiou, D., Asanovic, K.: RAMP: research accelerator for multiple processors. IEEE MICRO **27**(2), 46–57 (2007)
Weiss, S., Smith, J.E.: IBM Power and PowerPC. Morgan Kaufmann, San Francisco (1994)
Wikipedia: Linear programming [online] (2008). Available from: http://en.wikipedia.org/wiki/Linear_programming
Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem – overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. **7**(3), 1–53 (2008). doi:10.1145/1347375.1347389
Wolf, W.: The future of multiprocessor systems-on-chips. In: Proceedings of the 41st DAC '04, pp. 681–685 (2004)
Xilinx: Xilinx resource page [online] (2010). Available from: http://www.xilinx.com
Zhang, H., Ferrari, D.: Rate-controlled static-priority queueing. In: INFOCOM '93, Proceedings of the Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies, Networking: Foundation for the Future, pp. 227–236. IEEE, San Francisco (1993). doi:10.1109/INFCOM.1993.253355

Index

A. Kumar et al., *Multimedia Multiprocessor Systems*, Embedded Systems,
DOI 10.1007/978-94-007-0083-3, © Springer Science+Business Media B.V. 2011